Einführung in PASCAL

aus:

Happy Computer -- PASCAL / FORTH / C
Sonderheft 5/1986

Markt & Technik

Programmier-sprachen — Babel läßt grüßen

Wer den Computer zu seinem Hobby oder Beruf macht, steht in mehrerer Hinsicht vor einer schwierigen Wahl. Er muß sich zwischen einer Vielzahl von Peripherie, Betriebssystemen und Programmiersprachen entscheiden. Hier zeigen wir Ihnen die wichtigsten Programmiersprachen, ihre Entwicklungsgeschichte und was sie können.

Kommunikation braucht Sprache. Wenn sich zwei Menschen miteinander unterhalten, benutzen sie dazu das gesprochene Wort oder, sofern sie nicht die gleiche Sprache verstehen, Hände und Füße. Dieses Prinzip läßt sich nicht ohne weiteres auf die Verständigung zwischen Mensch und Computer anwenden. Kommunikation ist der Datenaustausch zwischen mehreren Parteien, die einander verstehen müssen, jedoch ist ein Computer im Urzustand alles andere als verständig. Er besitzt lediglich eine mehr oder weniger große Anzahl von Fähigkeiten, die er sehr schnell ausführen, aber nicht selbständig koordinieren kann.

Befehl und Gehorsam

Programmiersprachen sind, im Gegensatz zur weitverbreiteten Meinung, kein Mittel, um sich mit dem Computer zu unterhalten. Sie dienen lediglich dazu, dem Computer über eine Kette von Befehlen mitzuteilen, was er Schritt für Schritt zu tun hat. Programmierung wurde lange Zeit unter dem Hauptaugenmerk der Mensch-Maschine-Kommunikation betrachtet. Der wichtigste Aspekt lag darin, zu Problemlösungen unter möglichst effizienter Nutzung der Maschinenkapazität zu gelangen. Seit Computer in jüngster Zeit eine stärkere Verbreitung erfahren haben, rückte jedoch ein zweiter Gesichtspunkt immer mehr in den Vordergrund: Die Programme werden komplexer und der Wartungsaufwand (Korrektur oder Erweiterung eines Programms) immer größer. Dadurch, daß an vielen Programmen große Teams über lange Zeiträume hinweg arbeiten, geraten Programmiersprachen auch mehr und mehr zum Kommunikationsmittel zwischen diesen Gruppen von Menschen. Das bedeutet, daß ein guter Programmierer immer seine Programme auch für seine Nachwelt verständlich gestalten, und seine Kunstfertigkeit nicht mit der Anwendung von Programmiertricks unter Beweis stellen sollte. Einem potentiellen Benutzer, der das Programm lesen und eventuell ändern muß, sollte die Funktionsweise möglichst schon beim Lesen klar werden.

Statt »Programmiersprache« wäre die Bezeichnung »Kommandosequenz« eigentlich richtiger. Was dem Computer befohlen wird, führt er geduldig und beliebig oft aus, einzige Bedingung ist ein fehlerfreies Programm. Einige Psychologen behaupten denn auch, die Beliebtheit von Computern sei darauf zurückzuführen, daß viele Menschen ihre diktatorischen Triebe beim Programmieren ausleben!

Wenden wir uns der Frage zu, die wohl jeden Computeranwender irgendwann einmal bewegt, nämlich, was denn welche Programmiersprache wohl zu leisten vermag.

Im Laufe der Computergeschichte, die seit den ersten Relaisrechnern Konrad Zuses nicht mehr als ein halbes Jahrhundert zählt, wurde eine Unzahl Programmiersprachen, Spracherweiterungen und Dialekte entwickelt. Dabei standen immer zwei Überlegungen im Vordergrund. Zum einen mußten die Hardwarevoraussetzungen berücksichtigt werden. Zum anderen orientieren sich die Programmentwickler an den zu bearbeitenden Problemstellungen und den Bedürfnissen der Anwender. Je nach Computertyp und Problemstellung werden vom Programmierer verschiedene Programmstrukturen (Module, Blockkonzept, Verbundtypen), spezialisierte Befehle und unterschiedliche Datentypen (Integer, Real, Complex, String, etc.) benötigt. Auf dieser Grundlage entwickelten sich neben den bekannten Sprachen eine unüberschaubare Anzahl Exoten, die meistens nur in einzelnen Universitäten eingesetzt wurden.

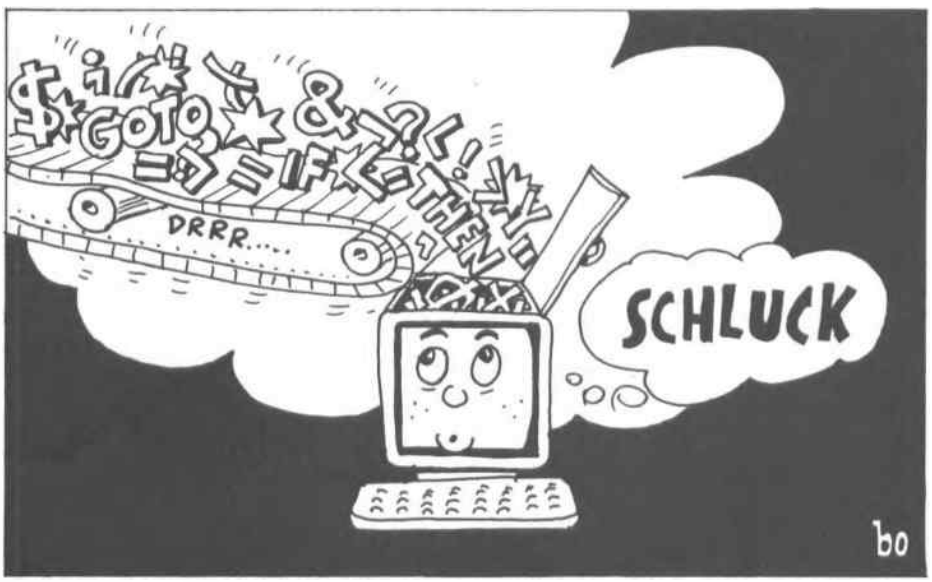
Vielfalt — Freud oder Leid?

An kaum einem Punkt scheiden sich die Geister in der Computerszene so stark wie in der Auswahl der Beurteilung der Programmiersprachen. Wer den Heimcomputer zu seinen Hobbies zählt, lernt in aller Regel zunächst fleißig Basic. Schließlich gehört ja der Basic-Interpreter zum Lieferumfang. Im Laufe der Zeit werden die eigenen Programme immer länger und unübersichtlicher, die einzelnen Programmteile sind durch ein unentwirrbares Geflecht von GOTO-Anweisungen miteinander verknotet (böse Zungen sprechen deshalb von »Spaghetti-Code«).

Mit dem wachsenden Bedürfnis nach Strukturierung, die in Basic nur jemand mit viel Selbstdisziplin erreicht, und nach Geschwindigkeit, die für Basic ein Fremdwort ist, sieht man sich nach Auswegen um. Dabei fühlt sich die eine Gruppe wie magisch von dem Begriff Assembler angezogen und begibt sich auf die unterste Sprachebene, um fortan in mühseliger Kleinarbeit Byte für Byte zu programmieren. Hiermit wird zwar ein Höchstmaß an Geschwindigkeit möglich, aber die Übersichtlichkeit kommt nach wie vor zu kurz. Die zweite Gruppe der Programmier-Gemeinde wendet sich deshalb modernen Hochsprachen zu, wie Pascal, Forth, Modula, Comal, Ada, C und so weiter. Diese Sprachen zwingen den Programmierer dazu, seine Programme modular (dies bedeutet, einzelne Aufgaben werden in »Paketen« oder »Modulen« zusammengefaßt) zu gestalten. Durch die so erreichte Übersichtlichkeit lassen sich Programme später von jedermann, Sprachkenntnisse vorausgesetzt, nachvollziehen und ändern. Zudem

sind einige dieser Sprachen sehr assemblernah, wie zum Beispiel Forth, womit auch Geschwindigkeit kein Problem mehr ist. Moderne Programmiersprachen unterstützen also Programmstrukturen und gehen ebenso mit Daten- und Kontrollstrukturen problemgerecht um.

Dennoch werden im professionellen Bereich (Universitäten, Verwaltung) »klassische« Sprachen bevorzugt, wie PL/1, Cobol und Fortran. Es sprechen auch gute Gründe dafür: So sind die neueren Programmiersprachen oftmals auf der vorhandenen Hardware noch nicht verfügbar, oder sie vertragen sich mit bereits vorhandenen Softwarekomponenten nicht. Ein anderer Faktor sind die Vorkenntnisse des Wartungspersonals und und und .... Jeder Informatikstudent, jeder Praktiker kann diese



Reihe beliebig fortsetzen. Und schließlich ist gute Software immer noch der Triumph des Programmierers, nicht der der Programmiersprache.

Seit das Betriebssystem CP/M auf Computern wie dem Commodore 128, Schneider CPC, oder dem Atari ST einen neuen Frühling erlebt, werden diese Klassiker neuerdings auch auf Heimcomputern interessant. Sehr wahrscheinlich wird sich ein breiter Anwenderkreis hierfür finden. Welcher Programmierer begrüßt es nicht, wenn er seine Produkte teilweise in der »guten Stube« austesten kann?

Eine Sonderstellung unter den Programmiersprachen nehmen die Sprachen für »künstliche Intelligenz« (KI) ein. Deren bekannteste Vertreter heißen Lisp und Prolog. KI-Sprachen bauen auf einem grundlegend neuen Konzept auf, bei dem der Programmierer dem Computer im Dialog sein Problem mitteilt. Wegen des hohen Speicherbedarfs sind KI-Programme auf Microcomputern nur sehr eingeschränkt einsatzfähig. Dies wird sich jedoch bald ändern. Man darf gespannt sein, wie sich KI auf den 16-Bit-Computern entwickeln wird, die ja nicht nur in punkto Schnelligkeit, sondern auch im Speicherangebot einen ganz neuen Standard setzen. Der Künstlichen Intelligenz ist in diesem Sonderheft ein eigener Beitrag gewidmet.

Fassen wir zusammen: Sinnvoll lassen sich die Programmiersprachen in vier Gruppen unterteilen:

1. Assemblersprachen: Sie bieten den Vorteil, daß sie die Möglichkeiten der Hardware optimal ausnutzen. Assembler versteht jeder Prozessor unmittelbar. Jede höhere Programmiersprache muß beim Programmablauf erst in Assemblercode übersetzt werden und ist deshalb weniger universell. Gegen Assembler sprechen die mühselige Programmierung, die Gebundenheit an Prozessor und Hardware sowie die schwierige Wartung.

2. Klassische Hochsprachen: Sie sind zur Zeit am weitesten verbreitet, aber genügen wegen ihrer altertümlichen Konzeption den Anforderungen moderner Programmierung nicht mehr.

3. Moderne Hochsprachen: Sie können sich gegen die »Alteingesessenen« nur langsam durchsetzen. Sie bieten Strukturierungshilfen, ausgeprägte Möglichkeiten der Datenbeschreibung, und unterstützen die Selbstdokumentation des Programmtextes durch eine angemessene Verbalisierung.

4. KI-Sprachen sind heutzutage noch Gegenstand intensiver Forschungen und stellen eine völlig neue und andersartige Klasse von Programmiersprachen dar. In diesem Beitrag wollen wir uns deshalb nicht weiter mit ihnen beschäftigen.

Wie Sie gesehen haben, ist der Sprachenwald immer noch sehr dicht, auch wenn man sich auf die Auswahl der wichtigsten Programmiersprachen beschränkt. Ein bedeutendes Auswahlkriterium sind die qualitativen Merkmale. Bevor wir diese besprechen, noch zu einigen zentralen Begriffen:

Algorithmus – Jedem Programm liegen ein oder mehrere Lösungsverfahren zugrunde, oft auch als Algorithmus bezeichnet. Ein solcher Algorithmus ist definiert als eindeutige und vollständige Vorschrift zur Lösung einer Problemklasse mit einer Abfolge von Schritten, die in einem endlichen Zeitraum ausgeführt werden.

Daten heißen dabei die Objekte, die der Algorithmus bearbeitet.

Programm nennt man folgerichtig die maschinengerechte Aufbereitung der Daten und des Algorithmus. Diese Aufbereitung geschieht mit Hilfe einer künstlichen Sprache, eben der **Programmiersprache**. Diese setzt sich aus einer Menge von Zeichen zusammen, die ihrerseits nach bestimmten Regeln zusammengesetzt werden können. Die somit geschaffenen Sprachelemente werden von der Maschine unmittelbar (als Maschinensprache) oder unter Zuhilfenahme von Übersetzungsprogrammen (als Hochsprache) »verstanden«. Der Übersetzer stellt nichts weiter dar, als einen speziellen Algorithmus, der in der Lage ist, alle Befehle einer Hochsprache in den entsprechenden Assemblercode zu transformieren. Er ist grob vergleichbar mit einer Bibliothek von kleinen Assembler-Unterprogrammen, wobei jedem Befehl der Hochsprache eines dieser Unterprogramme zugeordnet ist.

Compiler und Interpreter

Übersetzungsprogramme gliedern sich in zwei Typen, die die gleiche Aufgabe auf unterschiedliche Weise erfüllen. Als erstes sind die **Compiler** zu nennen. Sie tauchten auch in der geschichtlichen Entwicklung zuerst auf. Der Compiler übersetzt den Programmtext (Quellcode) in einem oder mehreren Durchgängen (Passes) komplett in Assemblercode (Objektcode). Der Compiler selbst wird daher beim Programmlauf nicht mehr benötigt. Natürlich benutzt man für unterschiedliche Computer, mit verschiedener Hardware und Maschinensprache auch unterschiedliche Compiler.

Der zweite im Bund der Dolmetscher nennt sich **Interpreter**. Er ist der fleißigere von beiden. Zunächst muß der Quellcode direkt im Arbeitsspeicher des Computers abgelegt werden. Sodann beginnt das Interpretieren. Das heißt nichts weniger, als daß der Interpreter während des Programmlaufs Befehl für Befehl holen muß. Natürlich ist diese Vorgehensweise in höchstem Maße unökonomisch und langsam. Während ein Compiler die ersten drei der genannten vier Arbeiten nur genau einmal ausführen muß, beschäftigen sie den Interpreter bei jeder Programmwiederholung aufs neue.

Zu Beginn des Computerzeitalters, als Rechenzeit noch sehr teuer war, wurden daher ausschließlich Compiler entwickelt. Interpreter konnten sich erst mit höheren Prozessorleistungen durchsetzen. Sie werden auch heute noch in Profikreisen wegen ihres gemächlichen Arbeitstempos verschmäht.

Die Qualität von Programmiersprachen

Um beurteilen zu können, welche Eigenschaften gute Programmiersprachen charakterisieren, wenden wir uns zunächst der Frage zu, welche Anforderungen an ein hochwertiges Programm zu stellen sind.

Die Forderung nach Fehlerfreiheit erscheint auf den ersten Blick banal. Die Erfahrung zeigt nämlich, daß 100%ig korrekte Programmsysteme ab einem gewissen Umfang kaum noch möglich sind. Hat ein Programm eine gewisse Komplexität erreicht, ist es fast unmöglich, seine Korrektheit zu beweisen, also Testdurchläufe zu finden, die tatsächlich alle Eventualitäten berücksichtigen. Als Maß für die Korrektheit eines Programmes wird deshalb häufig von der Zuverlässigkeit gesprochen. Diese gibt die Wahrscheinlichkeit an, mit der ein Programm für eine Zahl von Anwendungsfällen in einer bestimmten Zeitspanne fehlerfrei arbeitet. Gerade die jüngste Vergangenheit hat hierfür im Bereich der Mikrocomputer einige negative Beispiele geliefert. So konnten Fehler in einigen Betriebssystemen oftmals erst nach der Auslieferung der neuen Geräte beseitigt werden.

Verständlichkeit deckt sich mit Forderungen nach Lesbarkeit, Überschaubarkeit und Selbstdokumentation. Programme sollten sich, sobald sie eine gewisse Länge überschreiten, in funktionelle Einheiten gliedern. Andernfalls wird der Programmierer oft, noch während er an ein und demselben Programm arbeitet, an der selbst produzierten Unordnung scheitern. Man differenziert unter dem Aspekt der Verständlichkeit zwischen der statischen und der dynamischen Programmstruktur. Die statische Strukturgestaltung dient dem Ziel, ein übersichtliches Layout des Programmtextes zu gestalten. Hiermit wird der menschlichen Wahrnehmung Rechnung getragen, die sehr stark auf optischen Wahrnehmungen beruht. Ergänzend trägt die dynamische Strukturierung dazu bei, daß Programmabläufe unmittelbar aus dem Quelltext ersichtlich werden.

Ein Faktor der an diese Zusammenhänge anknüpft, ist die Änderbarkeit von Programmen. Wartungsfreundlichkeit bedeutet einerseits, daß Programme leicht an modifizierte Aufgabenstellungen angepaßt werden können. Andererseits spielt die Portabilität eine Rolle, wenn ein Programm zum Beispiel in einer neuen Softwareumgebung lauffähig gemacht werden soll.

Universalität sollte es einem guten Programm ebenfalls ermöglichen, ähnliche Aufgabenstellungen und Abwandlungen zu lösen.

Im Zusammenhang mit der Benutzerfreundlichkeit sollten Programme einen Dialog mit dem Benutzer ermöglichen und Eingabefehler abfangen, ohne falsche Ergebnisse oder gar Systemabstürze zu liefern.

Effizienz hat im Laufe der Entwicklung stark an Bedeutung verloren. Zu Zeiten, da Speicherplatz noch Mangelware war, galten die kürzesten Programme als die besten. Das ging natürlich zu Lasten der Übersichtlichkeit. Effizienzstreben wird heute daher nur noch mit Skepsis betrachtet.

Die hier genannten Qualitätsmerkmale stehen offensichtlich in positiver und negativer Wechselwirkung zueinander. So geht Übersichtlichkeit meistens zu Lasten der Effizienz, verbessert dagegen aber die Änderbarkeit.

Beginnen wir jetzt damit, die Anforderungen an eine ideale Programmiersprache zu formulieren. Die Qualität einer Computersprache läßt sich danach beurteilen, inwieweit sie die Entwicklung von Programmen unterstützt, die den uns bekannten Anforderungen entsprechen. Die folgenden skizzierten Merkmale müssen im engen Zusammenhang miteinander betrachtet werden.

Beginnen wir mit einem Punkt, der besonders den Einsteiger interessieren wird:

Die Erlernbarkeit einer Sprache hängt wesentlich von deren Struktur und Umfang ab. Sie ist sehr viel einprägsamer, wenn die Zahl der Schlüsselwörter gering und das Sprachkonzept durchgängig ist. Ebenso gewährleistet eine einfache Benutzung, wenn der Programmierer sein Problem bequem mit einer breiten Palette von Ausdrucksmöglichkeiten lösen kann.

Die Einheitlichkeit ist ein ebenso wichtiger wie unscharfer Begriff. Er soll im großen und ganzen bedeuten, daß für eine bestimmte Leistung möglichst nur genau ein Sprachmittel zur Verfügung steht.

Kriterien für eine ideale Sprache

Kompaktheit steht mit Einfachheit in Einklang. Hierbei ist nicht von der »Würze der Kürze« die Rede, sondern vielmehr von der Mächtigkeit der Sprachkonzepte. Der Bauplan der Sprache muß eine geringe Anzahl verschiedener Grundkonzeptionen aufweisen, wie mathematische Operationen, Ein- und Ausgabe, Datenstrukturierung. Andererseits soll Kompaktheit ein gewisses Maß an Redundanz (= alles was man in der gleichen Sprache auf andere Weise auch darstellen kann) an der richtigen Stelle nicht verhindern. Dies fördert die Verständlichkeit und Zuverlässigkeit. Beispielsweise sind Datentypen im Grunde genommen redundant, doch wer möchte sich schon mit einer Sprache herumschlagen, die ausschließlich den Datentyp »Zeichen« kennt?

Die Sprache Basic wird von vielen als katastrophal eingestuft. Das hat einen guten Grund: Die Forderung nach Lokalität wird von Basic so gut wie gar nicht unterstützt. Mit diesem Begriff ist gemeint, daß die Teile einer bearbeiteten Aufgabe, die logisch zusammengehören, auch im Programmtext in physischer Nachbarschaft stehen sollten. Die berühmt-berüchtigten Sprungbefehle bewirken jedoch das genaue Gegenteil.

Ein ganz anderes Kriterium ist die Sicherheit der Programmiersprache. Dazu zählt das Unterstützen der Fehlerfreiheit eines Programmes durch Sprachelemente. »On Error Goto« ist ein solcher weitverbreiteter Befehl. Des weiteren sind Sprachelemente zu nennen, die die Testphase des Programms fördern.

Der »Wildwuchs« der Implementierungen (= Anpassungen eines bestimmten Computers) bei Programmiersprachen, insbesondere bei den älteren, ist hinlänglich bekannt. Dialekte, Erweiterungen, aber auch Einschränkungen beeinträchtigen die Portabilität von Programmen im besonderen Maße. Mit der Standardisierung

Fortsetzung auf Seite 10

befassen sich daher nationale und internationale Institute. Die wichtigsten sind das Deutsche Institut für Normung (DIN), das American National Standards Institute (ANSI) und die International Organization for Standardization (ISO). Die Benutzer akzeptieren allerdings die Standards unterschiedlich. So sind Standards von Basic nahezu unbekannt, während sie sich bei Sprachen wie Pascal oder Fortran zunehmend durchsetzen.

Was für die Effizienz bei Programmen gesagt wurde, verkehrt sich bei den Sprachen in das genaue Gegenteil: Für den Übersetzer ist sie von großer Wichtigkeit. Zum einen sollte der Übersetzungsvorgang effizient sein, da bei der Fertigstellung eines Programmes die Zahl der Testläufe meistens sehr hoch ist. Der wichtigere Grund aber ist, daß das erzeugte Maschinenprogramm im Hinblick auf Rechenzeit und Speicherbedarf optimiert werden sollte. Sogenannte »Optimierende Übersetzer« sind teilweise in der Lage, das erzeugte Maschinenprogramm effizienter zu gestalten, als dies der Benutzer durch Programmiertricks erreichen kann.

Von Zuse bis Ada

Wir besitzen nun ein gutes Handwerkszeug, um Programmiersprachen nach den wichtigsten Gesichtspunkten theoretisch zu beurteilen. Fassen wir zusammen: Die entscheidenden Anforderungen an eine Sprache heißen Strukturierungshilfen, Selbstdokumentation, Datenbeschreibung, Benutzerfreundlichkeit und Zuverlässigkeit. Doch was nützt all die graue Theorie, wenn wir die Sprachen nicht kennen? Im folgenden werden daher die wichtigsten unter den bisher behandelten Aspekten und im Rahmen ihrer geschichtlichen Entwicklung vorgestellt.

Die Entstehung der Programmiersprachen orientiert sich immer auch an den Voraussetzungen der Hardware. Dies gilt insbesondere auch für die Gründerjahre.

Als geistiger Urvater der Rechenmaschinen mit Programmsteuerung darf der Engländer Charles Babbage gelten. Er begann 1833 mit der Konstruktion digitaler Rechenautomaten. Er legte seinen Maschinen aus Zahnrädern, Kurbeln und Hebeln zwei wichtige Erfindungen zugrunde, nämlich die Lochkartensteuerung und das Prinzip eines dekadischen Zählrades mit automatischem Zehnerübertrag. Babbages Projekte waren aber wegen fertigungstechnischer Schwierigkeiten nur in der Theorie funktionsfähig. Erst Elektromechanik und später die Elektronik machten die Rechenautomaten langsam zu Computern.

Der erste Rechenautomat der Welt mit Programmsteuerung wurde 1941 von Konrad Zuse in Betrieb genommen. Die Zuse Z3 war ein Relaisrechner, der bereits mit Dualzahlen arbeitete und zur Darstellung Gleitkommazahlen benutzte. Mit der Z3 waren neben den vier Grundrechenarten auch das Ziehen von Quadratwurzeln und das Potenzieren möglich. Ein Nachbau des historischen Modells, das im Krieg zerstört wurde, steht im Deutschen Museum in München. Der erste programmierbare Rechner Amerikas entstand 1944. Er wurde von dem Mathematiker Howard H. Aiken mit Unterstützung von IBM entwickelt und auf den Namen »Mark I« getauft. Er war jedoch ein Ungetüm von 16 m Länge und 35 Tonnen Gewicht und zudem langsamer als die früher entwickelte Z3.

Der Phase der Relaisrechner setzte der Einsatz von Elektronenröhren rasch ein Ende. Der bekannte ENIAC war die erste vollelektronische Rechenanlage der Welt und wurde 1945 in den USA fertiggestellt. Er erreichte gegenüber den Relaisrechnern bereits die 2000fache Rechengeschwindigkeit.

Während die Lochkarte nur eine starre Programmsteuerung ermöglichte (keine Schleifen, keine logischen Entscheidungen) begann man bald, sich über die flexible Speicherprogrammierung Gedanken zu machen. Als erstem gelang es dem Amerikaner John Neumann das genannte Problem auf einem Rechner zu verwirklichen. Der bereits 1944 von Neumann konzipierte Computer (EDVAC) erfüllte folgende Forderungen: Das Programm mußte, wie auch die zu verarbeitenden Daten, in der Maschine gespeichert werden. Außerdem benötigte man bedingte Befehle wie Vorwärts- und Rückwärtsverzweigungen. Jeden Befehl konnte zudem die Maschine selbst, wie jeden anderen Operanden ändern.

Befehle bestanden aus einem Operations- und einem Adreßteil. Im Operationsteil wird eine Angabe gemacht was zu tun ist (zum Beispiel Ausführung einer Multiplikation), der Adreßteil zeigt an, wo sich die zu verarbeitenden Daten befinden und wohin sie anschließend zu übertragen sind. Hier ist also die Rede von den ersten Assemblersprachen, an deren Grundprinzipien sich bis heute nicht geändert hat.

Der Schritt von der starren Programmsteuerung zum flexiblen Programm leitete die Wende vom Rechner zur Datenverarbeitung ein. Die Röhrencomputer der ersten Generation erreichten mit dem SSEC (Selective Sequenz Electronic Calculator) Ende der vierziger Jahre einen Höhepunkt. Dieser besaß nicht weniger als 12000 Elektronenröhren und etwa 21500 Relais und wurde von 36 Lochstreifenlesern gesteuert. Er führte die Berechnungen der Mondbahn durch, die 20 Jahre später im Apollo-Raumfahrtprogramm verwertet wurden. Mit dem Einzug der Transistortechnik und später mit den integrierten Schaltkreisen wuchsen fortan Rechnerleistung und Speicherkapazität immer schneller. Dies war die Voraussetzung für die Schaffung der höheren Programmiersprachen.

Die frühen Jahre — Fortran

Fortran ist die älteste der hier behandelten Hochsprachen und setzt einen Meilenstein in der Geschichte. Anfang der fünfziger Jahre wuchs die Zahl der Computer rasch. An Serienfertigung war noch nicht zu denken und es war jedes Gerät ein Einzelstück mit eigener Hardware und eigenem Assembler. So wurde bald der Wunsch nach einer Programmiersprache laut, die übertragbar und einfach zu programmieren sein sollte. 1952 wurde der Grundstein für Fortran gelegt, zu einer Zeit, da die Programmierung nur wenigen Spezialisten und ausschließlich in Assembler möglich war. John W. Backus war einer der Federführenden, dem die Programmiergemeinde Fortran zu verdanken hat.

Der Hauptgrund für die Entwicklung war die Schwerfälligkeit der Assemblerprogrammierung. 75 Prozent der Kosten eines Rechenzentrums verursachte damals die Fehlersuche. Verständlichkeit war daher ein wesentliches Entwurfsziel. Dadurch, daß die teuere Hardware optimal ausgenutzt werden mußte, waren die Rahmenbedingungen für Fortran bereits vorgezeichnet. Vorrangig wurden Sprachelemente implementiert, die der Speicher- und Laufzeiteffizienz nachkamen. Einige dieser Konzepte werden noch heute als sehr nachteilig angesehen – sind aber immer noch in Fortran enthalten.

1955 erschien ein Programmierhandbuch, und zwei Jahre später wurde die erste Implementierung auf einer IBM 704 freigegeben. Damit stand Fortran erstmals einer breiten Zahl von Programmierern zur Verfügung.

Der Name steht für FORmula TRANslating system (Formelübersetzer). Und genau dort liegt auch der Anwendungsschwerpunkt der Sprache. Rechnerische Probleme lassen sich in ihr leicht und natürlich ausdrücken. Damit wird

der Erlernbarkeit der Sprache Rechnung getragen. Im ingenieurwissenschaftlichen und mathematischen Bereich gilt Fortran auch heute noch als die wichtigste Programmiersprache. So bietet sie zum Beispiel neben den allgemein gebräuchlichen Zahlentypen Real (Fließkommazahlen) und Integer (Ganze Zahlen) auch noch den Typ »Double Precision« für Rechnungen mit höherer Genauigkeit sowie »Logical« für boolsche Operationen. Großrechner-Versionen beinhalten zudem noch den Typ »Complex«, der in der theoretischen Elektrotechnik eine sehr wichtige Rolle spielt.

Das Format dieser Sprache ist streng zeilenorientiert und erlaubt normalerweise nur einen Befehl je Zeile. Das hängt damit zusammen, daß Fortran zunächst als lochkartenorientierte Sprache entstand. Grundsätzlich mußte man damals für jede neue Anweisung eine neue Lochkarte (beziehungsweise Zeile) verwenden. Wie auch in Basic, das später aus Fortran entstand, mußte bei den ersten Versionen viel mit dem Goto-Befehl umhergesprungen werden. Neuere Versionen wie Fortran V und Fortran 77 bieten demgegenüber schon strukturierende Sprachelemente wie IF....THEN....ELSE...ENDIF.

Nachdem die 1958 geschaffene Version Fortran II eine mäßige Verbreitung gefunden hatte, entstand 1962 das in weiten Kreisen akzeptierte Fortran IV. Den fortschreitenden Auswüchsen immer neuer Versionen wurde 1966 Einhalt geboten, mit einer Version, die größtenteils mit Fortran IV identisch war. Schließlich überarbeitete das ANSI Fortran 66 im Jahr 1977 nochmals und beseitigte einige eklatante Mängel.

Unter CP/M ist Fortran derzeit für fast alle Mikrocomputer mit Z80-Prozessor erhältlich, ebenso wie eine Reihe von Fortran-Implementationen für MS-DOS-Computer.

Cobol — die Geschäftige

Die Programmiersprache Cobol entstand 1959 auf Initiative des US-Verteidigungsministeriums. Zu dieser Zeit begann Fortran sich gerade auszubreiten. Was noch fehlte, war eine Sprache für den kommerziellen und kaufmännischen Einsatz. So entwickelte man Cobol mit dem Ziel, große Datenbestände verarbeiten zu können und die Ein/-Ausgabe zu unterstützen. Insbesondere die ersten Fortran-Versionen waren hierfür ungeeignet. Ende der fünfziger Jahre wurde die Codasyl-Entwicklungsgruppe aus Vertretern der Computerindustrie und der amerikanischen Regierung gegründet. Schon 1960 stellte diese Gruppe die erste Version mit der Bezeichnung Cobol-60 vor. Sie war wesentlich an die weniger bekannte Sprache Comtran (COMmercial TRANslator) angelehnt. Aufgrund der hastigen Entwicklung von Cobol innerhalb eines halben Jahres ergaben sich viele Ungereimtheiten, die sich teilweise durch alle Neuentwürfe hindurchschleppten und auch heute noch nicht ganz beseitigt sind. Cobol-61 war dann die Grundlage für alle späteren Versionen. Sie war zu Cobol-60 nicht kompatibel. 1965 wurde als wesentliche Neuerung die Unterstützung von Massenspeichern und Tabellen mit eingebracht.

Die Sprachelemente von Cobol sind je nach ihrer Funktion in Module zusammengefaßt. Das ANSI entwickelte bis 1974 einen zwölf Module umfassenden Standard, namentlich Cobol ANS-74. Er wurde 1980 nochmals verbessert. Den jeweils neuesten Stand veröffentlicht das CODASYL-Komitee im Abstand von drei Jahren. Der Standard für die Bundesrepublik Deutschland ist in der DIN-Norm 66028 nachzulesen.

Die kurze Darstellung der Entwicklungsgeschichte läßt erkennen, daß Cobol ebenfalls eine alte Sprache ist. Cobol-Programme müssen aus heutiger Sicht als mangelhaft angesehen werden.

Ursprünglich verfolgte man wie bei keiner anderen Sprache das Ziel der Lesbarkeit des Programmtextes. Die Sprache sollte dann auch leicht erlernbar sein. Der Programmtext erinnert stark an (englische) Prosa. Cobol-Programme simulieren nämlich die natürliche englische Sprache. So wird zum Beispiel jeder Befehl mit einem Verb eingeleitet. Die Grundrechenarten stehen nicht als Symbole, sondern als Befehlswörter (add, divide) zur Verfügung. Ebenso werden logische Operatoren ausgeschrieben. Ein Beispiel: Wenn die Variable A größer ist als Null, soll der Variablen B die Summe der Variablen C und A zugeordnet werden. In Basic würde man das so formulieren:

```
IF A>0 THEN B= C+A
```

Daraus wird in Cobol:

```
IF A GREATER THAN ZERO ADD C TO
A GIVING B.
```

Daß so aus komplizierteren mathematischen Formeln monströse Gebilde werden, leuchtet ein. Da höhere mathematische Funktionen in Cobol ganz fehlen, ist die Sprache für wissenschaftliche Anwendungen völlig ungeeignet.

Derartige Beispiele verdeutlichen, daß das Entwicklungsziel von Cobol nicht sinnvoll erreicht wurde. Dennoch ist Cobol auch heute noch die weltweit am stärksten verbreitete Programmiersprache. Ganze Rechenzentren arbeiten mit ihr. Die hohen Investitionen und die Gewöhnung des Personals an diese Sprache wird auch in Zukunft für ihren Erhalt sorgen. Zudem erreicht auch noch keine neuere Programmiersprache die Cobol-Domäne Datenorganisation.

Wer einen Mikrocomputer mit den Betriebssystemen CP/M oder MS-DOS (PC-DOS) besitzt, kann in Cobol einsteigen.

PL/1 — von allem etwas

Fortran und Cobol sind charakteristisch für die strikte Trennung in kommerzielle und technisch-wissenschaftliche Anwendungen zu Beginn der sechziger Jahre. Daneben wurden Computer nur noch in Spezialgebieten eingesetzt. Im Laufe der Zeit traten aber die Merkmale der naturwissenschaftlich-technischen Bereiche in den betriebswirtschaftlichen Anwendungen immer mehr hervor und umgekehrt. So waren die kommerziellen Anwender mehr und mehr auf Methoden aus der Statistik, Operations Research und Ökonomie angewiesen. Mathematiker und Ingenieure stellten zunehmend höhere Ansprüche an Datenverwaltung und an die Ein-/Ausgabeunterstützung.

Als logische Konsequenz kamen die Anbieter den neuen Bedürfnissen mit einer universellen Hard- und Softwarekonfiguration nach. Hardwareseitig entwickelte IBM die Rechnerfamilie /360 mit dem Betriebssystem OS/360. Diese Anlagen zählten sich bereits zur dritten Computergeneration (das heißt, die Schaltkreise waren in Hybridtechnik ausgelegt, einer unmittelbaren Vorstufe der integrierten Schaltkreise).

Bei den Überlegungen zu einer neuen Sprache war man zunächst von Fortran ausgegangen. Die Organisation SHARE (Society for Help to Avert Redundant Effort), eine Vereinigung wissenschaftlicher IBM-Anwender, einigte sich mit der Firma IBM auf die Gründung eines Sprachkomitees. Zunächst war die Rede von Fortran VI. Man gelangte aber schon nach kurzer Zeit zu der Erkenntnis, daß die gewünschten Verbesserungen eine Kompatibilität mit Fortran unmöglich machten. Die Anlehnung an Fortran hätte außerdem die große Gruppe der kommerziellen Verwender abgeschreckt. So entschied man sich für die Entwicklung einer gänzlich neuen Sprache. Deren wichtigsten Entwurfsprinzipien waren: allgemeine Einsetzbarkeit und weitgehende Ausdrucksfreiheit. Der Sprachaufbau sollte modular sein und Testhilfen sowie Möglichkeiten zur

Fehlerbehandlung bieten. Ein großenteils gegenläufiges Ziel bestand in den Forderungen, den vollen Zugriff auf Hardware- und Betriebssystemleistungen bei gleichzeitiger Maschinenunabhängigkeit zu gewähren.

Nach vielen drastischen Überarbeitungen hatte sich der Sprachumfang bis 1965 stabilisiert. Nachdem die Abkürzung für NPL (New Programming Language) bereits vergeben war, einigte man sich schließlich auf PL/I (Programming Language one). Im August 1966 wurde dann der erste Compiler für eine IBM /360 freigegeben. Schließlich verabschiedeten das ANSI und die ECMA, ein europäisches Standardisierungskomitee, einen vorläufig endgültigen Standard. Die Verbreitung von PL/1 nahm zunächst rasch zu, wurde aber später den gesetzten Erwartungen nicht gerecht.

Kommen wir nun zu der Sprache, die jeder, und wenn nur vom Hörensagen, kennt. Sie kann sich rein zahlenmäßig im Mikrocomputerbereich als die am

In den Sprachumfang von PL/1 wurden viele Konzepte aus Fortran, Cobol, Algol und Jovial übernommen (letztere werden hier wegen ihrer geringen Verbreitung nicht behandelt). Leider gelang es nicht, sich bei der Auswahl nur auf die guten Eigenschaften der Vorgänger zu beschränken. Zudem ist der Sprachumfang riesig. Eine fast unüberschaubare Anzahl von Schlüsselwörtern zwingt zu Maßnahmen, durch die der Programmierer auch mit Teilmengen der Sprache sinnvoll arbeiten kann. Daher ergeben sich für die PL/1 -Syntax sehr freizügige Vorschriften. Einerseits existiert keine Zeilenstruktur, Zwischenräume, Einrücken und Kommentare dürfen fast beliebig verstreut werden. Da passiert es dann auch nicht selten, daß beim Programmieren das Format aus allen Fugen gerät. Außerdem sind die Schlüsselwörter nicht reserviert, wohl wegen ihrer großen Anzahl. Gebilde wie

```
IF ELSE=THEN THEN IF=ELSE; ELSE
IF=THEN
```

tragen wohl zur Verwirrung jedes hoffnungsvollen Programmierers bei. Ein wesentlich angenehmerer Fortschritt ist andererseits das ausgeprägte Blockkonzept der Sprache. PL/1 orientiert sich wesentlich an Prozeduren. Allgemein kann gesagt werden, daß die Einarbeitung in PL/1 viel Zeit braucht. Wer damit trotz aller Warnungen beginnen will, kann dies unter CP/M oder auf PC-Kompatiblen tun.

weitesten verbreitete Sprache der Welt rühmen. Sie ist schon fast als Teil der Allgemeinbildung zu betrachten. Ganz anders als bei den »großen professionellen« wurde die Entwicklung von Basic (Beginners All-purpose Symbolic

Instruction Code) nicht durch eine Industrie- oder Militärlobby getragen. Die Ausrichtung der Sprache kommt denn auch in ihrem Namen zum Ausdruck: Sie wendet sich an den Anfänger und soll für jeden Zweck geeignet sein.

Basic — Basis für Einsteiger

Ihre geschichtliche Entwicklung erklärt viele Aspekte des Sprachkonzepts. Basic wurde von Thomas Kurtz und John Kemeny von 1956 bis 1971 in den USA am Dartmouth College entwickelt. Ziel war es, Studenten, die sich nicht ausschließlich mit Ingenieurwissenschaften beschäftigten, das Programmieren zu erleichtern. So schlugen sich denn auch die Erfordernisse der allgemeinen Ausbildung einer Universität in der Sprache nieder: Bei der angesprochenen Zielgruppe erschien ein eigener Programmierkurs nicht erforderlich. Vielmehr sollte das Programmieren im Rahmen der Mathematikvorlesungen gelehrt werden. Hieraus erklärt sich auch die Ausrichtung von Basic auf mathematische Probleme. Die neue Sprache sollte leicht erlernbar und leicht zu benutzen sein. Dies war nach Meinung von Kurtz und Kemeny bei Fortran und Algol nicht der Fall. So erklärt sich auch, daß bei Basic nicht, wie bei PL/1, auf Bewährtes zurückgegriffen wurde.

Im Gegensatz zu den bisher behandelten Sprachen wurde Basic als vollständiges Programmiersystem konzipiert. Der Benutzer kann im Dialog mit Basic arbeiten, ohne die Basic-Umgebung zu verlassen. Hierzu existiert der »Direkt-Modus«, der das Editieren, Ausführen und Speichern von Programmen unterstützt. Basic ist zudem eine typische Interpreter-Sprache, für die allerdings auch verschiedene Compiler erhältlich sind.

Mit der Entwicklung des ersten Compilers begannen Kurtz, Kemny und eine Gruppe Studenten 1963. Im Mai 1964 wurde dann das erste Basic-Programm ausgeführt; die erste Version kannte nur 14 Instruktionen. Diese Minimalausstattung wurde am Dartmouth College bis 1971 in insgesamt sechs Versionen schrittweise vervollständigt. Seither nahmen die Autoren keine Veränderungen mehr vor. Das bedeutet natürlich nicht, daß Basic jemals eine wirkliche Standardisierung erfahren hätte. Im Gegenteil: Durch das Fehlen einer Interessenvertretung professioneller Anwender und durch die enorme Verbreitung der Sprache wucherten die Basic-Versionen fast uferlos. Für nahezu jeden Mini- und Mikrocomputer und selbst auf Taschenrechnern ist Basic erhältlich. Da von Herstellerseite nach dem Motto verfahren wurde »jedem Topf ein anderer Deckel« ist Kompatibilität ein Fremdwort. Auch die Bemühungen der ECMA und ANSI in den letzten Jahren waren kaum von Erfolg gekrönt. Lediglich ein »Minimal Basic« wurde kompromißbereit zum Standard deklariert. Diese Teilmenge ist der ersten Sprachbeschreibung von 1964 sehr ähnlich. Lediglich im Heimcomputerbereich gelang es mit den MSX-Computern erstmals, einen weitestgehenden Basic-Standard für Geräte verschiedener Hersteller zu schaffen. Haar in der Suppe war aber, daß die MSX-Computer wenig Verbreitung fanden.

Ein Basic-Programm besteht aus nummerierten Zeilen in aufsteigender Folge. Dabei sind die Zeilennummern aus einem festgelegten Intervall zu wählen. Jeder Zeilennummer folgen eine oder mehrere Instruktionen. Die

Zeilennummern legen die logische Reihenfolge der auszuführenden Anweisungen fest. Darüber hinaus dienen sie als Orientierung für die Sprungbefehle. Beim Editieren des Programmtextes lokalisieren sie die Zeilen. Basic ist noch stärker zeilenorientiert als Fortran. Das geht so weit, daß eine Basic-Anwendung durch die Zeilenlänge begrenzt wird. In einigen Basic-Versionen sind Fortsetzungszeilen mit dem Zeichen »&« möglich.

Als Datentypen sind in Basic nur numerische Daten und Zeichenketten vorgesehen. Eine Unterscheidung in ganzzahlig und Fließkomma, wie es von anderen Sprachen her bekannt war, wurde der Einfachheit wegen bewußt vermieden, ist aber dennoch auf vielen Computern implementiert. Für Datenstrukturen stehen nur ein- und mehrdimensionale Felder (Arrays) zur Verfügung. Deren Elemente können je nach Version entweder nur einzeln manipuliert werden, oder es stehen spezielle Matrizenoperationen bereit.

Die Sprunganweisungen sind, wie bereits erwähnt, vielen ein Greuel. Zwar bieten moderne Basic-Versionen viele Befehle, die das strukturierte Programmieren unterstützen, sie erfordern aber genaue Vorausplanung eines Programms. Im Regelfall endet bei längeren Programmen ein »Drauflosprogrammieren« im Chaos aus GOTO, GOSUB und FOR...NEXT.

Des weiteren beinhaltet Basic, je nach Ausstattung und Computer, Befehle für die Ein-/Ausgabe, mathematische Funktionen, Grafik, Tonerzeugung und den Zugriff auf das Betriebssystem. Hierauf im einzelnen einzugehen würde zu weit führen. Es sei daher auf entsprechende Fachliteratur verwiesen.

Pascal - strukturiert und einfach

Ein moderner Klassiker unter den Programmiersprachen ist Pascal. Der Name ist ausnahmsweise keine Abkürzung. Er wurde zu Ehren des französischen Mathematikers Blaise Pascal gewählt, der 1642 im Alter von 19 eine der ersten funktionsfähigen Rechenmaschinen konstruierte.

Die Wurzeln dieser Programmiersprache reichen in das Ende der sechziger Jahre zurück. Zu jener Zeit stellten Nikolaus Wirth und C.A.R. Hoare Überlegungen an, auf der Basis von Algol 60 eine Nachfolgesprache zu entwickeln. Algol 60 bot schon damals ein zufriedenstellendes Sprachkonzept. Das gilt besonders für die Strukturierung des Programmablaufs und des Programmtextes. Wegen dieser Vorteile wurde Algol zur Grundlage einer ganzen Klasse von Programmiersprachen, die bei Pascal beginnt und vorläufig von Ada gekrönt wird. Eine der Schwächen ist dagegen die unzureichende Datenstrukturierung. Ebenso wie Basic und Fortran kennt Algol 60 nur das Array. Die später folgende Version Algol 68 war ähnlich wie PL/1 zu umfangreich und unhandlich. Mit der Entwicklung von Pascal verfolgte man das entgegengesetzte Ziel. Der Schweizer Professor Nikolaus Wirth formulierte bei der Entwicklung der Sprache an der ETH Zürich die folgenden Schwerpunkte: Pascal sollte nur grundlegende Sprachkonzepte enthalten. Diese sollten natürlich definiert sein und das Erlernen des strukturierten Programmierens als eine systematische Disziplin unterstützen. Des weiteren sollte sie sich effizient auf allen Computern implementieren lassen.

Die erste vorläufige Version entstand 1968. Die vollständige Beschreibung eines Compilers und der Sprache selbst war 1971 fertig. Das 1974 erschienene Benutzerhandbuch »Pascal User Manual and Report« enthält eine Sprachdefinition, die heute als Wirth-Standard bezeichnet wird. Die verbreiteten Versionen Turbo- und UCSD-Pascal enthalten demgegenüber noch einige Erweiterungen, die vor allem Grafik und Zeichen-Strings betreffen.

Durch die leichte Erweiterbarkeit von Pascal entstanden bald viele Dialekte. Deshalb und wegen der weltweiten Anerkennung der Sprache nahmen sich verschiedene Normeninstitute diesem Problem an. Die ISO setzte schließlich 1980 einen Standard fest, der in der DIN-Norm 66256 nachzulesen ist. Strukturierung bedeutet nicht nur, daß das Programm übersichtlich ist, sondern daß sich der Vorgang des Programmierens in verschiedene Aktionen aufteilt. Bevor man sich an den Computer setzt, sollte man das Problem genau analysieren und in Aufgabenpakete zerlegen. Dann ist für jedes Paket ein Algorithmus zu bestimmen und in Pascal zu formulieren. Erst dann beginnt die Tipparbeit. Pascal-Programme entstehen also auf dem Papier und weniger am Bildschirm. Diese Vorgehensweise erreicht eine niedrige Fehlerquote und damit sinkt auch die Zahl der Übersetzerdurchläufe, was bei einer typischen Compilersprache wie Pascal sehr angenehm ist.

Ein Pascal-Programm ist klar gegliedert in einen Vereinbarungs- und einen Anweisungsteil. Vereinbart werden zuerst alle Variablen, Konstanten und deren Datentypen. Im Anweisungsteil werden die Aufgabenpakete in Procedures formuliert. Jede Procedure enthält einen eigenen Namen. Das eigentliche Hauptprogramm besteht dann nur noch aus dem Aufrufen der Procedures und steht im Programmtext ganz am Ende.

Daß der GOTO-Befehl in Pascal enthalten ist, verwundert eigentlich. Angesichts der Strukturbefehle IF....THEN-CASE kann man gut auf ihn verzichten. Der Vorrat an Datentypen eröffnet gegenüber Basic ganz neue Möglichkeiten. Man unterscheidet hier zwischen einfachen Typen, strukturierten Typen und Zeigertypen. Integer, Char, Boolean und Real zählen zu den einfachen Typen. Boolean bezeichnet eine Variable, der nur die Werte False oder True zugeordnet werden können. Array, Record, Set und File stehen als strukturierte Typen zur Verfügung. Set bezeichnet eine Menge. Auf diesen Typ sind die üblichen Mengenoperationen Vereinigung, Durchschnitt, Differenz, Untermenge und Elementüberprüfungen anwendbar. Record ermöglicht Verbundvariablen. Es lassen sich so unterschiedliche Variablentypen zu einer Variable zusammenfassen. Record war ursprünglich für kommerzielle Anwendungen gedacht (Tabellendarstellung). Demgegenüber werden mit dem Typ File nur Variablen eines einzigen Typs verkettet. Der Typ Zeiger (pointer) schließlich ermöglicht verkettete Listen und deren bequeme Manipulation, sowie Baumstrukturen. Wem das noch nicht reicht, der kann sich in Pascal weitere einfache Datentypen selbst definieren.

Pascal ist vielseitig und erzieht zum strukturierten Denken. Seine Verarbeitung, vor allem im akademischen Bereich, ist folgerichtig sehr hoch. So liegen denn auch für fast alle rechnereigenen Betriebssysteme wie auch für CP/M und MS-DOS Pascal-Versionen vor.

Forth – die etwas andere Sprache

Forth entstand Anfang der siebziger Jahre. Charles H. Moore entwickelte die Sprache ursprünglich zur Steuerung von Radioteleskopen. Er arbeitete dazu mit einem IBM-1130, einem Computer der dritten Generation. Das Endprodukt war aber so mächtig, daß es Moores Computer als einen der vierten Generation erscheinen ließ. Er wollte der neuen Sprache daher den Namen Fourth geben. Namen mit mehr als fünf Buchstaben waren auf dem IBM jedoch nicht erlaubt. So wurde das »u« ein Opfer dieser technischen Unzulänglichkeit.

Forth ist interaktiv wie Basic. Das heißt, es existiert sowohl ein Interpreter als auch ein Compiler. Programme können somit erst im Direktmodus »häppchenweise« getestet und anschließend compiliert werden. Des weiteren verbindet Forth Merkmale der Assemblersprachen mit denen der Hochsprachen.

Die Strukturierung in Forth entsteht durch die Definition immer neuer Worte. Der ohnehin schon große Sprachumfang nimmt beim Programmieren ständig zu. Beliebige Befehle können zu einem neuen Befehl zusammengefaßt werden, der dann sofort wieder in weitere Befehle mit eingebaut werden kann. Schließlich steht für das gesamte Programm ein einziger Befehl am Ende dieser Kette.

Selbstverständlich stellt Forth auch die Kontrollstrukturen zur Verfügung, die bereits für Pascal angesprochen wurden, wie IF.....ELSE....ENDIF, BEGIN...UNTIL, BEGIN...WHILE etc. Das berüchtigte GOTO fehlt hier ganz, ergäbe auch bei diesem Sprachkonzept keinen Sinn.

Grundlegendes Prinzip von Forth ist das Operieren mit dem Stapelspeicher (Stack). Dieser funktioniert nach dem LIFO-Prinzip (Last In, First Out). Alle Werte, die auf dem Stapel abgelegt wurden, lassen sich nur in umgekehrter Reihenfolge wieder herunternehmen. Für einen problemlosen Ablauf dieses Systems sorgen eine ganze Reihe von Stack-Befehlen, mit denen sich Werte verschieben und vertauschen lassen. Sämtliche mathematischen Operationen laufen in Forth über den Stack. Man bedient sich hierbei der Umgekehrten Polnischen Notation (UPN), die recht gewöhnungsbedürftig und Benutzern von HP-Taschenrechnern bekannt ist.

Forth ist ein sehr offenes System und durch seine Assemblernähe universell einsetzbar. Fehlende Funktionen können jederzeit selbst programmiert werden. Zudem ist Forth sehr schnell. Das Erlernen der Sprache und die Übersichtlichkeit der Programme können als noch ausreichend eingestuft werden. Auf jeden Fall fasziniert Forth jeden, der sich länger damit beschäftigt. Für alle verbreiteten Mikrocomputer existieren mittlerweile eine oder mehrere Versionen.

Logo — kinderleicht

Seymour Papert, der als geistiger Vater der Sprache Logo gilt, arbeitete 12 Jahre an der Verwirklichung dieser »Erziehungsphilosophie«. Er leitete ein eigens gegründetes Entwicklungsteam aus Programmierern und Lehrkräften am MIT (Massachusetts Institute of Technology) in den USA. Man arbeitete damals ausschließlich auf den größten vorhandenen Datenverarbeitungsanlagen. Dadurch fand ein wesentliches Konzept der KI-Sprache Lisp in Logo Anwendung: die Listenprogrammierung. Listen sind einfach zu definieren und können per Befehl manipuliert, kombiniert und verglichen werden. Eine leicht programmierbare Dateiverwaltung ist nur ein Anwendungsbeispiel dieser Technik.

Bekannt wurde Logo vor allem durch die Schildkröte, ein kleines Zeichensymbol der »Turtle-Graphics«. Mit ihr lassen sich auf einfache Weise die tollsten Grafiken zaubern. Die Schildkröte krabbelt über den Bildschirm und hinterläßt dabei eine sichtbare Spur.

Eine Schildkröte machte Logo bekannt

Gesteuert wird mit einfachen Befehlen wie FORWARD, BACK, LEFTTURN, RIGHTTURN. Zusätzlich muß noch die Länge der zurückgelegten Strecke und des Drehwinkels angegeben werden. Ebenso ist eine Standortabfrage der Turtle möglich. Logo-Programme ähneln in der Struktur dem Baukastenprinzip der Forth-Programme. Mit Hilfe des Interpreters lassen sich einzelne Bausteine erproben und später zum eigentlichen Programm zusammensetzen. Der Komfort ist dabei in Logo ungleich höher als in allen bisher genannten Sprachen. So kann der Anwender vorerst Begriffe wie Archivierung, Dateien und andere spezielle Funktionen der Datenverarbeitung links liegen lassen. Diese Vorteile gehen aber leider zu Lasten des Speicherplatzes.

Eng mit dem Prozedurkonzept verbunden ist die Rekursivität von Logo. Prozeduren sind in der Lage, sich selbst aufzurufen. Auf diese Weise lassen sich schnell reizvolle grafische Gebilde erzeugen und gewisse mathematische Zusammenhänge einfach ausdrücken. Rekursive Strukturen sind in Basic gar nicht und in vielen anderen Sprachen nur mit hohem Aufwand zu verwirklichen.

In die Logo-Philosophie wurden Erziehungstheorien des Schweizer Philosophen Jean Piaget eingebracht. Dieser hatte zuvor das Lernverhalten von Kindern analysiert. Tatsächlich wirkt Logo besser auf die Denkweise eines Schülers als Basic oder Pascal. Bemängelt werden muß bei Logo hauptsächlich die geringe Verarbeitungsgeschwindigkeit der Programme. Sie fällt aber bei einem Lernsystem nicht so stark ins Gewicht.

Wegen des hohen Speicherbedarfs sind Logo-Interpreter auf Mikrocomputern nur als Ausschnitt des Gesamtsystems erhältlich. Dies wird sich mit wachsendem Speicherstandard jedoch bald ändern.

Comal — gelungene Essenz

Im Jahre 1973 ging Comal aus den Sprachen Basic und Pascal als ein neuer Ableger hervor. Später kamen in Comal noch Elemente von Logo hinzu, so zum Beispiel die Schildkrötengrafik. Zudem sind in Comal der Compiler und der Interpreter nicht getrennt vorhanden, sondern es wurden deren beste Elemente in einer Zwischenstufe zusammengefaßt. Ein Comal-Programm besteht aus drei Schritten. Im ersten wird schon bei der Programmeingabe die Syntax überprüft. Dieser Syntaxchecker ist selbst für den ohnehin schon eingabefreundlichen Interpreter ungewöhnlich komfortabel. Die Comal-Schlüsselworte werden sofort in sogenannte »Token« übersetzt, das sind Abkürzungen, die nur ein Byte beanspruchen. Dieses Prinzip verwenden übrigens alle Interpreter. Der zweite Schritt beginnt nach dem Programmstart. In einer Art Compilerdurchlauf wird der Programmtext nach Variablen, Prozeduren, Funktionen und Sprüngen durchsucht. Die Ergebnisse dieser Analyse werden dann in einer gesonderten Liste zusammengefaßt. Man kann diesen Vorgang auch als eine Art automatische Deklaration ansehen. Im dritten Schritt, dem Programmlauf selbst, wird auf diese Liste ständig direkt zugegriffen. So ergeben sich gegenüber Basic, wo der Interpreter oft den ganzen Programmtext absucht, enorme Geschwindigkeitsvorteile.

Die Comal-Syntax lehnt sich stark an die von Basic an. Im Direktmodus wurden die meisten Befehle Wort für Wort übernommen. Das gleiche gilt für einige Befehle des Programm-Modus. Einige Comal-Versionen akzeptieren neben den eigenen Schlüsselworten sogar noch gleichbedeutende Basic-Befehle. Wer aus Basic zu Comal aufsteigt, ist so zwar vor Irrtümern einigermaßen sicher, jedoch wird dem Prinzip der Eindeutigkeit nicht gerade Rechnung getragen.

Von Pascal wurde die Strukturiertheit übernommen, dessen strenge Syntaxvorschriften aber erfreulicherweise vermieden. Die typischen Kontrollstrukturen, die schon bei Pascal und Basic beschrieben wurden, sind ausnahmslos vorhanden. Die Lesbarkeit des Pro-

grammtextes wird zudem dadurch unterstützt, daß Comal beim Listen eine optische Gliederung vornimmt.

Comal wird für fast alle gängigen Mikrocomputer angeboten. Erfreulicherweise stehen auch, ähnlich wie bei Forth, zahlreiche Public-Domain-Versionen zur Verfügung. Diese unterscheiden sich von kommerziellen Sprachangeboten lediglich im Umfang. Es wird erwartet, daß Comal in Zukunft eine starke Verbreitung erfährt. Jüngster Anhaltspunkt für diese These ist der Beschluß der Kultusministerien, Pascal im Informatikunterricht allmählich durch Comal zu ersetzen.

C – die Zukunft?

»C« – für diesen einen Buchstaben lassen viele Programmierer Pascal, Forth oder Fortran links liegen. Viele betrachten C als die Programmiersprache schlechthin. Tatsächlich bietet C viele bestechende Vorteile, die sich von denen der anderen Hochsprachen unterscheiden. In C wurden bekannte Betriebssysteme wie Unix und GEM geschrieben.

Die Entwicklung von C reicht bis an den Anfang der siebziger Jahre zurück. Die Namensgebung war ebenso kurios wie einfallslos. 1970 begann die Firma Digital Equipments mit der Entwicklung von Spezialsprachen, die die Programmierung von Minicomputern unterstützen sollten. Diese wurden einfach nach dem Alphabet benannt. B war also eine Weiterentwicklung von A und diente 1971 dazu, Unix auf verschiedene Rechner zu übertragen. Bald darauf erkannten Dennis Ritchie und Ken Thompson, die damals bei Bell Laboratories beschäftigt waren, die Leistungsfähigkeit von B. Sie verbesserten diese Sprache bis 1973 entscheidend. Wie das Endprodukt heißt, wissen wir bereits. Unix wurde wenig später ebenfalls auf C umgeschrieben.

Wenn ganze Betriebssysteme in C gehalten sind, läßt sich die ungeheuere Effizienz dieser Sprache schon erahnen. Das C-Compilat geht mit dem Speicher äußerst sparsam um, und ist zudem um den Faktor 50 schneller als vergleichbare Basic-Programme.

Der eigentliche Befehlsvorrat von C ist denkbar klein. Er umfaßt nur 13 Instruktionen. Bei der Erzeugung eines C-Programms wird der Quelltext zuerst mit dem mitgelieferten Editor oder einer Textverarbeitung geschrieben. Dieser dient dann als Eingabedatei für den Compiler. Bis hierher besteht also kein Unterschied zur Vorgehensweise mit den meisten anderen Compilersprachen. Der Compiler überprüft den Text und übersetzt diesen mit den ihm bekannten Befehlen in eine Zwischendatei und legt diese auf einen externen Speicher (zum Beispiel Diskette) ab. Anschließend beginnt das sogenannte »Linking«. Der Linker ordnet aus einer externen Bibliothek die noch fehlenden Befehle entsprechendem Assemblercode zu. So entsteht schließlich das lauffähige Programm, das fast so kompakt ist, als wäre es direkt in Maschinensprache geschrieben. Das Compilieren benötigt jedoch wegen des Zwischencodes je nach Geschwindigkeit des Speichermediums mehr Zeit, als die unmittelbare Übersetzung in Maschinensprache.

Die Vorteile des Compiler-Linker-Prinzips wiegen diesen Nachteil bei weitem auf: C ist praktisch uneingeschränkt portabel und die Linker-Funktion kann man selbst erweitern. C-Programme bieten neben der selbstverständlichen Strukturierung einige ungewöhnliche aber vorteilhafte Eigenschaften. So sind in Anlehnung an Assembler Befehle zum Inkrementieren und Dekrementieren vorhanden, Variable können als Registervariable deklariert werden. Daß die Benutzung derartiger Sprachmittel einem Compiler das Leben leicht macht, ist klar. Einschränkungen entstehen jedoch bei Prozessoren, die nur wenige Register aufweisen, wie beispielsweise die drei 8-Bit-Register des 6502, der nur A, X und Y zu bieten hat.

Ein weiteres nennenswertes Stilmittel unter C sind Makros. Diese sind mit Unterprogrammen vergleichbar, die nicht angesprungen werden müssen, sondern jeweils erneut vom Compiler in den Objektcode eingebunden werden. Diese Technik bringt einen großen Zeitvorteil, da Parameterübergaben und Sprünge entfallen.

Im 8-Bit-Bereich ist C nur auf dem Z80 unter CP/M verbreitet. Eine Version für den C 64 stellt hier eine erfreuliche Ausnahme dar. Für 16- und 32-Bitter existieren aber umfangreichere Versionen.

Ada – gekrönter Adel

Ada ist heute als Krönung bei der Entwicklung modularer Programmiersprachen anzusehen. Die Sprache wurde erst in jüngster Zeit im Auftrag des weltweit größten Softwaresponsors entwickelt, dem Pentagon. Der finanzielle und organisatorische Aufwand dafür war entsprechend riesig. Benannt ist die Sprache nach der jungen Gräfin Ada Byron, die um 1830 für Babbages (siehe oben) Rechenmaschinen ein nahezu komplettes Programm zur Berechnung der Bernoullischen Zahlen schrieb.

Der Aufwand, der um Ada getrieben wurde, erklärt sich mit einer Kalkulation des US-Verteidigungsministeriums. Danach können zwischen 1983 und 1999 etwa 24 Milliarden Dollar (!) eingespart werden, wenn eine einzige universelle Programmiersprache die bisherigen 450 (!!) Programmiersprachen ersetzen könnte. Ada ist ähnlich PL/1 sehr umfangreich. Es wäre daher zwecklos, auf einzelne Sprachelemente einzugehen. Deshalb hier nur die grundsätzlichen Sprachkonzepte von Ada:

- Das Modulkonzept von Ada ist äußerst umfangreich. Es stehen sowohl datenorientierte als auch funktionsorientierte Module zur Verfügung. Innerhalb der datenorientierten Pakete lassen sich fast beliebige Datentypen und Datenstrukturen realisieren.
- Ähnlich wie in Pascal können Datenstrukturen geschachtelt werden. Umfangreichere Prozedur- und Funktionskörper werden ausgelagert, zum Beispiel um die Lesbarkeit der Programme zu erhöhen.
- Sämtliche Kontrollstrukturen, von UNTIL bis hin zu CYCLE-Schleifen stehen zur Verfügung. Ferner sind alle linearen und strukturierten Datentypen implementiert.
- Ein automatischer Textformatierer (Pretty-Printer) wertet Schachtelungsstrukturen aus und sorgt für ein übersichtliches Layout des Programmtextes.
- Neben Parallelverarbeitung (Multitasking) gehören Konzepte wie die Parallel- und Ausnahmebehandlung zu den bemerkenswerten Fähigkeiten, auf deren Erklärung hier wegen der komplexen Zusammenhänge verzichtet wird.

Im großen und ganzen wurden die gesetzten Ziele bei der Entwicklung von Ada nach dem heutigen Erkenntnisstand optimal erreicht. Die zu Anfang unter den Qualitätsaspekten genannten Stichworte wie Vollständigkeit, Zuverlässigkeit, Korrektheit, Übertragbarkeit, Wartung und Fehlerbehandlung wurden weitestgehend realisiert. Die Einfachheit der Sprache ist als ausreichend zu betrachten, wenn auch die vollständige Einarbeitung in dieses System Jahre beansprucht. Ada ist für Mikrocomputer zur Zeit nur unter MS-DOS verfügbar und auch hier nur in einer abgespeckten Version.

Wir sind nun mit dem Aufstieg im »modernen Turm zu Babel« fast an der Spitze angelangt. Nach unten blickend können wir die gebräuchlichsten Sprachen beurteilen.

(Matthias Rosin/ev)

Reifeprüfung in Pascal

Pascal gilt als hervorragend geeignete Computersprache für Lehre und Unterricht. Zwei Pascal-Systeme treten an, dem Atari ST auch den Schulbereich zu erschließen. Ist der ST reif fürs Abitur?

Computer sind von Natur aus vollkommen respektlos. Unbekümmert dringen sie in alle nur denkbaren Bereiche ein und zeigen selbst davor keine Scheu, in die geheiligten Gefilde des deutschen Schulwesens einzubrechen. Jedem Hüter humanistischen Bildungsgutes muß es wahrhaftig monochrom vor Augen werden, wenn sich seine ihm anvertrauten Zöglinge plötzlich mit einer seltsamen Sprache namens Pascal beschäftigen, statt wie gewohnt Latein oder Griechisch zu büffeln. Und das geschieht dann auch noch an diesen äußerst suspekten Computern.

Sprachbegabt

Doch letztlich muß auch eine so altehrwürdige Institution wie die Schule der modernen Zeit ihren Tribut zollen. Der Informatikunterricht und die Computersprache Pascal haben inzwischen einen festen Platz im Lehrplan erhalten. Die strenge Sprachstruktur und die Problemorientierung machen Pascal ohne jeden Zweifel zu einer idealen Sprache für diesen Anwendungsbereich. Dabei ist Pascal weit entfernt davon, nur eine akademische Lehrsprache zu sein. In Pascal kann man nämlich »richtig« programmieren und Programme erzeugen, die man verkaufen kann. Wer in der Schule Pascal lernt, hat also nicht nur für die Schule, sondern tatsächlich etwas für das Leben gelernt.

Der Atari ST hat sich in der kurzen Zeit seit seiner Markteinführung als ausgesprochen sprachbegabt erwiesen. Neben Logo, C, Basic (hier stottert er noch ein klein wenig), Forth, Modula und verschiedenen anderen Sprachen, spricht und versteht er seit einiger Zeit auch Pascal.

Auf dem Markt gibt es im Moment zwei Pascal-Systeme zu kaufen, das von Atari selbst vertriebene ST-Pascal (Preis: zirka 250 Mark) und das MCC-Pascal (Preis: 340 Mark) der englischen Firma Metacomco. Beide Pascal-Versionen stellen komplette Entwicklungssysteme dar, die auch die Programmierung der grafischen Bedieneroberfläche GEM zulassen. Allerdings sind beide Systeme TOS-Applikationen, also nicht in eine grafische Oberfläche eingebunden.

Zum Lieferumfang des ST-Pascal gehört neben einem 74-seitigen deutschsprachigen Handbuch in haltbarer Ringheftung eine 3½-Zoll-Diskette mit 23 Dateien. Der Besitzer eines Entwicklungspaketes Atari 520 ST findet hier einige alte Bekannte vor, nämlich die GEM-Libraries AESBIND und VDIBIND sowie die Hilfsprogramme BATCH.TTP, RM.PRG und WAIT.PRG aus dem C-Compiler-System von Digital Research. Der Pascal-Compiler besteht aus einer gut 128 KByte großen Datei PASCAL.PRG. Dazu gehören die speziellen Pascal-Libraries PASLIB und PASGRA.O, und eine Textdatei ERRORS.TXT mit den Standard-Fehlermeldungen des Pascal. Vervollständigt wird das Angebot durch einige Steuerdateien zum Compilieren und Linken (die .BAT-Dateien), durch den Linker FASTLINK.PRG, eine Include-Datei SCREEN.INC mit den VT52-Steuercodes, einige Beispielprogramme als Pascal-Quelltext und durch den TOS-Editor EDIT.TTP.

Der Editor ist ein Full-Screen-Editor mit Steuerung durch Cursortasten und Control-Codes. Die Funktionstasten sind nicht belegt. Ebenfalls vorhanden sind einige höhere Editor-Funktionen wie Suchen/Ersetzen und Blockverschieben. Nach Eingabe von CTRL-K können Befehle zur Steuerung des Datenverkehrs mit Diskettenlaufwerken eingegeben werden.

Dieser Editor erlaubt auch das Bearbeiten umfangreicher Quelltexte. Wer allerdings schon einmal einen Text-Editor mit grafischer Bedieneroberfläche benutzt hat, wird den GEM-Komfort schmerzlich vermissen. Da der Compiler des ST-Pascal ASCII-Dateien als Quelltexte erwartet, kann jedoch selbstverständlich jeder Editor benutzt werden, der solche Dateien erzeugt.

Im Gegensatz zu dem nicht allzu komfortablen Editor ist der Compiler PASCAL.PRG geradezu luxuriös ausgestattet. Er kann sowohl durch Anweisungen aus einer Kommandozeile beim Compilerstart, als auch durch Anweisungen aus dem Quelltext in vielfältiger Weise gesteuert werden.

Das Handbuch gibt ausführlich darüber Auskunft. Der Compiler erzeugt eine Objektcode-Datei im 68000-Maschinencode, die ein Linker (FASTLINK.PRG) mit Teilen der Library-Dateien zu fertigen Programmen verbindet. Diese Programme enthalten ein sogenanntes Runtime-Modul und sind deshalb wie andere TOS- oder GEM-Applikationen aus dem GEM-Desktop abrufbar. Der Sprachumfang nach dem ISO-Standard ist beträchtlich erweitert, zum Beispiel durch Implementierung des Variablentyp STRING. Weiterhin sind die sehr schnellen Grafik-Routinen des ST-Betriebssystems, das sogenannte LINE_A-Interface, direkt als Standard-Prozeduren und -Funktionen aufrufbar. Ebenso einfach erfolgt der Aufruf von Betriebssystem- und GEM-Routinen. Spezielle Externdeklarationen (GEMDOS, XBIOS, BIOS und C) im Quelltext bewirken die Einbindung der entsprechenden Library-Module beim Linken. Alle vordefinierten Funktionen erläutert das Handbuch recht ausführlich, sogar mit kleinen Beispielprogrammen. Dennoch kann das Handbuch ein Pascal-Lehrbuch für Anfänger nicht ersetzen. Hierzu sei auf die einschlägige Fachliteratur verwiesen.

Der schon erwähnte Linker FASTLINK.PRG ersetzt die beiden Programme LINK68.PRG und RELMOD.PRG, die mit den ersten Versionen des ST-PASCAL geliefert wurden. Die hohe Geschwindigkeit des Linkvorganges mit FASTLINK.PRG ist in Bild 1 dokumentiert. Besitzer des C-Compilers von Digital Research aus dem Entwicklungspaket können den neuen Blitzlinker auch für das Linken Ihrer C-Programme verwenden.

Schnell wie der Blitz

Bei allem Positiven, was bisher über das ST-Pascal zu vermerken war, darf ein Ärgernis nicht unerwähnt bleiben. Der Pascal-Compiler ist mit einem einfachen Kopierschutz versehen, der aufgrund seiner Primitivität kaum Schutz vor illegalem Kopieren bietet. Dafür bringt er aber beim Arbeiten mit einer RAM-Disk oder, wenn einmal verfügbar, mit einem Festplatten-Speicher unnötigen Zeitverlust beim Compilieren. Selbst wenn alle Dateien der Pascal-Systemdiskette auf eine RAM-Disk oder eine Festplatte übertragen sind, muß sich bei der Compilierung die Pascal-Diskette im Diskettenlaufwerk A befinden. Bei jedem Compilerlauf, egal auf welchem Externspeicher sich der Compiler befindet, wird mehrfach (!) auf

eine besonders präparierte Schutzspur auf der Pascal-Diskette zugegriffen. Dankenswerterweise hatte der Programmierer beim Linker ein Einsehen und verzichtete auf derartige Kindereien.

Das MCC-Pascal wird auf zwei Disketten geliefert, die insgesamt 57 Dateien enthalten. Das mitgelieferte Handbuch mit etwa 200 Seiten ist leider nur in englischer Sprache erhältlich. Wer Englisch gut beherrscht, findet hier allerdings sehr ausführliche Erläuterungen zu Befehlsumfang und Systembedienung. Das System setzt sich aus ähnlichen Elementen zusammen wie beim ST-Pascal. Man findet Editor, Compiler, Include-Dateien zum Aufrufen von GEM-Routinen, Linker und Libraries, sogar gut dokumentierte Assembler-Quelltexte der GEM-Libraries sind vorhanden. Auch hier trifft man zwei wohlbekannte Programme. Wie bei allen Metacomco-Computersprachen wird der TOS-Editor ED.TTP mitgeliefert. Dieser Editor von beträchlicher Leistungsstärke war bereits Bestandteil der zweiten Serie der Atari-Entwicklungspakete, und dürfte entsprechend weit verbreitet sein. Leider hat sich Metacomco noch nicht zur Einbindung in eine GEM-Oberfläche entschließen können.

Als Linker findet eine Updateversion des GST-Linkers aus dem GST-Assembler und dem GST-C Verwendung. Er kann hier aber nicht wie etwa im Assembler aus einer grafischen Bedieneroberfläche gestartet werden. Besitzer des GST-Assemblers können jedoch ohne große Schwierigkeiten das gesamte Pascal-System in die Oberfläche des Assemblers integrieren. Den Pascal-Compiler startet nämlich auch die Option »Run Program« im File-Menü. Der Linker läßt sich wie im Assembler aus dem entsprechenden Link-Menü steuern und starten. Auch die vielfältigen Funktionen zur Dokumentation des Linkvorganges sind ansprechbar.

Der Compiler PASCAL.TTP besitzt viele Optionen, die jedoch nur aus einer Kommandozeile bei Compilerstart, nicht aber aus dem Quelltext, aufrufbar sind. Er erzeugt aus dem Pascal-Quelltext eine Objektcode-Datei im GST-Format mit dem Dateityp .BIN, die durch den GST-Linker bearbeitet werden kann. Es ist aber auch eine Compiler-Option einstellbar, die die Compilierung des Quelltextes in das Format des ASS68 aus dem Entwicklungspaket ermöglicht. Aus diesem Grunde sind die Pascal-Library PASLIB, die GEM-Library GEMLIB und das Runtime-Modul STARTUP als .BIN-Dateien für den GST-Linker und als .O-Dateien für Digital Research Linker (LINK68 oder FASTLINK) auf den Disketten vorhanden.

Zwei Disketten voll Pascal

Zum Test und zum Vergleich mit dem ST-Pascal wurde nur die GST-Option verwendet. Die fertigen Programme sind wie beim ST-Pascal auch ohne Pascalsystem lauffähig und wie normale GEM- oder TOS-Applikationen zu benutzen.

Was leistet der MCC-Pascal-Compiler? Um es gleich vorweg zu sagen, MCC-Pascal ist nur etwas für ausgesprochene Pascal-Puristen. Der Sprachumfang umfaßt genau das ISO-Standardpascal, nicht mehr, aber auch nicht weniger. Der Variablentyp STRING fehlt ebenso wie die Funktion KEYPRESS. Dinge also, die die Pascalprogrammierung erleichtern. Leider kann die vorliegende Version des MCC-Pascal keine Betriebssystem- oder LINE_A-Aufrufe durchführen. Die GEM-Funktionen sind über eine Gruppe von INCLUDE-Dateien zugänglich, die EXTERN-Deklarationen von AES- und VDI-Routinen sind entsprechend den Digital Research-Spezifikationen zum GEM-System enthalten. Die eigentlichen Routinen befinden sich in der Library GEMLIB, die allerdings noch nicht ganz fehlerfrei zu arbeiten scheint.

Bei einem Vergleich der beiden Pascal-Versionen wurden die Zeiten für Compilierung und Linken des Benchmark-Klassikers »Sieb des Eratosthenes« sowie die Laufzeiten der erzeugten Programme für einen Durchlauf zur Ermittlung der Primzahlen gemessen (Ergebnisse in Bild 1).

Der MCC-Compiler arbeitet sehr schnell. Er ist beim Arbeiten mit der Diskette schneller als der ST-Pascal-Compiler mit einer RAM-Disk. Genau umgekehrt verhält es sich hinsichtlich der Linkdauer. Hier hat FASTLINK eindeutig die Nase vorn. Geradezu sensationell ist jedoch der Unterschied in der Laufzeit der Programme. Mit gut 17 Sekunden braucht das mit MCC-PASCAL erzeugte Programm länger als das entsprechende Turbo-Pascal-Programm unter CP/M-Emulation.

Bild 2 gibt einen weiteren Test wieder, der neben der Laufzeit der Programme die Rechengenauigkeit mit Fließkommazahlen überprüft. Dabei wird in einer WHILE...DO-Schleife von einer vorgegebenen Zahl solange der Wert 0,1 subtrahiert, bis die vorgegebene Zahl den Wert 0 unterschreitet. Dies müßte bei einem Startwert 100000 nach genau einer Million Substraktionen erreicht sein. Wie Bild 2 zeigt, besteht der ST-Pascal-Compiler diesen Tast mit Bravour. Erst bei Startwert 100000 treten merkbare Ungenauigkeiten auf. Der MCC-Pascal-Compiler macht bereits bei Startwert 100 sichtbare Fehler, bei Startwert 100000 wird der Wert 0 bereits nach 990564 Substraktionen erreicht.

Aufgrund dieser Ergebnisse fällt eine abschließende Beurteilung leicht. Das MCC-Pascal ist teurer, hat den geringeren Sprachumfang, erzeugt langsamere Programme und rechnet ungenauer. Auf der Plusseite lassen sich die geringere Compilierzeit und der bessere Editor verbuchen.

In allen wichtigen Punkten ist jedoch der ST-Pascal-Compiler die eindeutig bessere Wahl. Trotz des geringeren Preises besitzt er neben dem kompletten Sprachumfang des ISO-Standard-Pascal viele wichtige Erweiterungen, die die Fähigkeiten des Atari ST besser ausnutzen. Auch die Laufgeschwindigkeit der erzeugten Programme läßt kaum Wünsche offen. Die Abiturreife kann ohne Bedenken zuerkannt werden. (W. Fastenrath/hb)

Compilierzeiten »Sieb des Eratosthenes«

	Diskettenbetrieb		RAM-Disk		Laufzeit
	Compiler	Linker	Compiler	Linker	
MCC-PASCAL	12,7 sec	106,0 sec	1,9 sec	9,8 sec	17,3 sec
ST-PASCAL	51,0 sec	29,7 sec	14,8 sec	1,1 sec	0,7 sec

Bild 1. S wie super, T wie Tempo: Supertempo von ST-PASCAL

Rechengenauigkeit mit REAL-Zahlen

	MCC-PASCAL			ST-PASCAL		
Startwert	Endwert	Subtraktionen	Zeit	Endwert	Subtraktionen	Zeit
100	0,0010	1000	1,3	0,0000	1000	0,5
1000	0,0971	10000	7,6	0,0000	10000	3,0
10000	0,0491	100015	72,8	0,0000	100000	25,8
100000	0,0964	990564	660,6	0,0853	1000000	241,0
200000	-	-	-	0,0437	2000000	478,3

Bild 2. Rechnen will gelernt sein

Pascal auf dem C64

Pascal ist an Universitäten und in Software-Häusern gleichermaßen anerkannt. Probleme vollständig zu analysieren und strukturiert zu programmieren – das bringt Pascal Ihnen bei.

Die von Wirth entwickelte Sprache Pascal lief zunächst nur auf Großrechnern. Seinen Siegeszug auf Mikrocomputern trat Pascal an, als auf der Universität von California in San Diego (UCSD) das berühmte UCSD-Pascal entstand. Es ist eine Anpassung des Wirth'schen Standard-Pascal an einen Mikrocomputer und enthält sowohl Einschränkungen als auch Erweiterungen.

Pascal-Versionen für den Commodore 64 gibt es etwa seit 1983. Dabei handelt es sich in der Regel um »Enkel« des Wirth'schen Urcompilers, bei denen allen man gewisse Abstriche machen muß: Sie erzeugen keine Maschinensprache für den C64, sondern einen Zwischencode, auch P-Code genannt.

Der Compiler übersetzt in P-Code

Im Gegensatz zur Standard-Sprache Basic benutzt UCSD-Pascal einen Compiler, der den Pascal-Quellcode in P-Code übersetzt. Der P-Code (P steht für Pseudo) ist einer wirklichen Maschinensprache sehr ähnlich. Man kann sich diese Sprache als Modell eines Computers vorstellen. Es besitzt eigene Register, einen Stack, einen Heap für dynamische Variablen und einen Prozessor, der den P-Code ausführt. Da all dies die Software zuwege bringt, die Maschine also nur virtuell vorhanden ist, läuft P-Code um einiges langsamer als echter Maschinencode. Die Laufzeiten liegen dann um etwa zwei- bis fünfmal höher als bei echtem Maschinencode.

Man sollte aber die Vorteile dieses Verfahrens nicht übersehen. P-Code schlägt Basic in der Geschwindigkeit immer noch bei weitem (etwa 10 mal so schnell). Der erzeugte P-Code ist kompakter als der übliche Maschinencode und benötigt daher weniger Speicherplatz.

Gegenüber Standard-Pascal weist KMMM-Pascal einige Funktionen mehr auf. Dazu gehört vor allem eine komplette String-Behandlung. Auch Bit-Operationen auf Integer-Variablen sind erlaubt. Maschinensprache-Routinen können von KMMM aus aufgerufen werden. Peek und Poke sind ähnlich wie in Basic zu verwenden.

Das Programm besteht aus einem Editor, einem Compiler und einem Translator. Der Editor ist ein formatfreier Full-Screen-Editor mit überdurchschnittlichem Komfort (im Vergleich zum Basic-Editor). Er besitzt eine Funktion zur Überprüfung der Syntax und belegt alle Funktionstasten. Eine häufig benötigte Befehlssequenz kann als Makro definiert werden.

Der KMMM-Editor erzeugt eine sequentielle Datei, weshalb auch andere Editoren den Programmtext bearbeiten können. Danach wird der Compiler geladen. Er erzeugt bei Bedarf ein Listing auf dem Drucker und wandelt das Quellprogramm in P-Code um. Nach dem Laden des Translators übersetzt dieser nun den P-Code in schnelle Maschinensprache.

Oxford-Pascal unterstützt weitgehend alle Elemente von Standard-Pascal. Eine spezielle String-Verarbeitung unterblieb. Es verwendet statt dessen Variablen vom Typ PACKED ARRAY OF CHAR. Dafür stehen zusätzliche Befehle zum Erzeugen von Grafik und Sound bereit. Die Möglichkeiten des C64 lassen sich also auch von Pascal aus nutzen. Der Bildschirm kann sogar in einen Grafik- und in einen Textbereich aufgeteilt werden. Solche Programme laufen allerdings deutlich langsamer.

Unterstützt werden auch Peek und Poke, das Einfügen von Routinen in Maschinensprache und die Programmierung der eingebauten Uhr. Ärgerlich ist lediglich die zu dünn geratene Dokumentation von etwa 50 Seiten, bei der auch noch die Nummerierung der Seiten und ein Stichwortverzeichnis fehlen.

Die Entwicklung von kleineren Programmen gestaltet sich mit Oxford-Pascal angenehm. Der Basic-Editor ist lediglich etwas erweitert. Ansonsten arbeitet man genauso wie mit einem Basic-Programm. Der Compiler und das übersetzte Programm befinden sich gleichzeitig im Speicher. Allerdings fehlen in diesem Modus einige Befehle. Erst im Disk-Modus verfügt man über den gesamten Sprachumfang, muß aber dann die Zeit für das Laden des Compilers und das Speichern der Programme in Kauf nehmen.

Oxford-Pascal übersetzt in P-Code. Es gibt aber einen Locate-Befehl, der eine P-Code-Datei in eine von Basic aus ladbare Datei umwandelt. Eine solche Datei benötigt dann das Oxford-System nicht mehr.

Profi-Pascal hat gegenüber den obigen Pascal-Versionen einen entscheidenden Vorteil. Der Zugriff auf die 1541-Floppy geht wegen der zusätzlichen Diskettenroutinen dreimal so schnell vor sich. Allerdings wurde dieser Vorteil auch mit einem entscheidenden Nachteil erkauft. Profi-Pascal besitzt nämlich praktisch ein eigenes Betriebssystem mit Editor, Compiler und Hilfsprogrammen. Auf die Kompatibilität zu anderen Programmen und Dateien wird damit verzichtet.

Nach dem Laden erscheint ein Hauptmenü, von dem die anderen Programmteile (Editor, Compiler) aus geladen werden. Profi-Pascal enthält viele zusätzliche Routinen über den normalen Sprachumfang hinaus. So ist beispielsweise der Zugriff auf den gesamten Arbeitsspeicher auf sehr elegante Weise möglich. Der in Standard-Pascal nicht vorgesehene Typ String erlaubt die Manipulation von Zeichenketten. Der Direktzugriff auf Sätze innerhalb einer Datei ist mit der zusätzlichen Prozedur Seek möglich.

Als weitere Eigenschaften sind das Verketten von Dateien und das Segmentieren von Programmen zu nennen. Programme mit Segment-Prozeduren laden einen Teil des Codes während des Programmlaufs in den Arbeitsspeicher nach.

Profi-Pascal besitzt komfortable Möglichkeiten, Assembler-Code in das Programm einzubinden. Schließlich enthält es einen eigenen Assembler. Der Pascal-Code selbst wird allerdings nicht in Maschinensprache, sondern in P-Code übertragen. Nachteilig ist, daß der Compiler nur auf der (natürlich kopiergeschützten) Original-Diskette arbeiten kann.

Pascal für Einsteiger und Profis

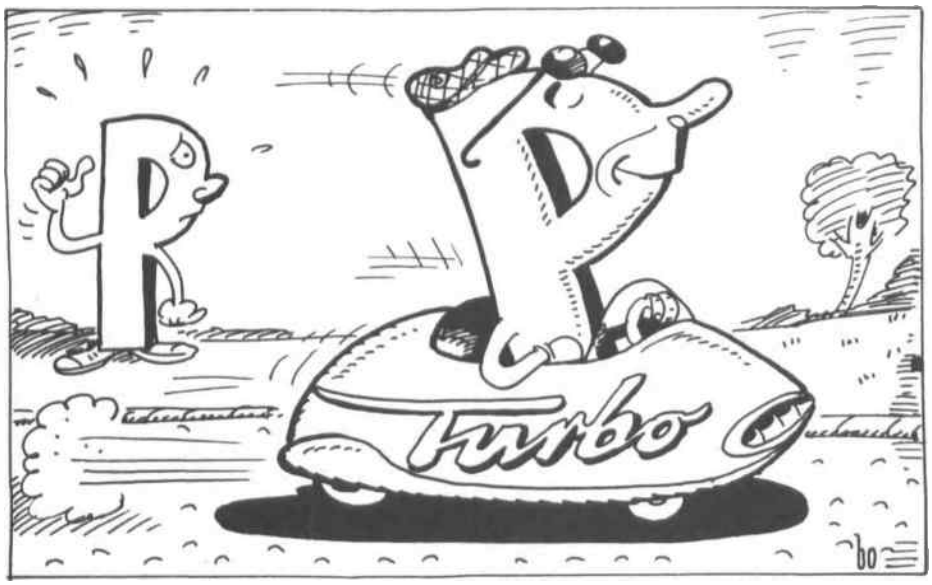
Eine Pascal-Version für den preiswerten Einstieg in die interessante Sprache ist ganz neu im Markt & Technik Verlag erschienen. Es handelt sich dabei um einen kompletten Pascal-Kurs in Buchform, der dazu sehr *schnell* und leistungsfähig ist und ohne Aufpreis dem Buch gleich auf Diskette beiliegt. Neben dem Compiler enthält die Diskette noch viele nützliche Beispielprogramme. Dieses »Pascal für den C 64« nutzt für seine Programme den gesam-

ten Speicher des C 64 aus. Ein Full-Screen-Editor ermöglicht eine komfortable Programmeingabe. Der Compiler akzeptiert den gesamten Standard-Sprachumfang mit einigen Erweiterungen. Das Buch enthält einen Einführungskurs mit vielen Beispielen und Übungsaufgaben für den Anfänger sowie Tips und Tricks für den Profi. Mit gutem Gewissen kann man Buch und Compiler als preiswerte Alternative zu den bisher bekannten Pascal-Versionen für den C64 empfehlen. Denn bei diesem Pascal-Compiler gibt es trotz des extrem niedrigen Preises keinerlei Abstriche an Sprachumfang oder Dokumentation. Als Autor von Buch und Compiler zeichnet Florian Matthes, den wir auch als Autoren für den Pascal-Kurs in diesem Sonderheft gewinnen konnten.

Die Entscheidung für einen der vier Compiler ist sicherlich nicht einfach. Wer einfache Bedienung wünscht, dem sei Oxford-Pascal empfohlen. Reine Maschinensprache wird nur von KMMM-Pascal erzeugt. Profi-Pascal wiederum bietet den größten Sprachumfang.

Und ein C 64-Fan, ob Einsteiger oder Profi, wird schließlich mit Markt&Technik-Pascal am meisten Spaß haben.

(Anton Gruber/cg)



Turbo-Pascal: der Renner

Seit etwas mehr als zwei Jahren gibt es einen Pascal-Compiler, der auf dem Computermarkt Furore gemacht hat: Turbo-Pascal. Was das Programm alles leistet, ist schon fast unheimlich.

Die Gründe für den grandiosen Erfolg von Turbo-Pascal sind recht einfach nachzuvollziehen. Da ist zuerst einmal der (damals) konkurrenzlos niedrige Preis. In den USA kostet Turbo-Pascal etwa 50 Dollar – in Deutschland um die 200 Mark. Ein Direktimport lohnt sich aber dennoch nicht, weil Sie Zoll und Verpackungsgebühren bezahlen müßten, die den scheinbaren Preisunterschied wieder ausgleichen.

Einen Pascal-Compiler für 200 Mark – das gab es vor Turbo-Pascal nirgends auf dem Computermarkt. So fanden sich nicht wenige, denen das Risiko eines Fehlkaufs klein genug erschien und sich aus diesem Grund Turbo-Pascal gekauft haben. MS-Pascal von Microsoft und Pascal/MT+ von Digital Research, die beiden vormals hauptsächlich benutzten Compiler, kosteten je nach Betriebssystem zwischen 1000 und 2000 Mark. Für MS-Pascal gilt dies heute noch, während Pascal/MT+ in einer Version für den Schneider CPC 6128 inzwischen auch auf ungefähr 200 Mark herunterging. Damit zielt es direkt auf den Turbo-Pascal-Markt.

Allein der Preis kann aber dennoch nicht für den Erfolg von Turbo-Pascal verantwortlich sein. Schließlich gibt es heute auch schon Pascal-Compiler ab 100 Mark (zum Beispiel Nevada-Pascal), die trotz ihres günstigen Preises eher ein Schattendasein führen.

Zählen wir also weiter Turbo-Pascal-Vorzüge auf: Turbo-Pascal ist ein integriertes Programmpaket, das aus einem Editor, dem eigentlichen Compiler und einer Laufzeitumgebung besteht. Was das heißt, kann nur jemand ermessen, der schon mit anderen Compilersprachen gearbeitet hat. Ein typischer Arbeitszyklus sieht dort etwa so aus: Texteditor laden (meistens wird keiner mitgeliefert, also muß Wordstar oder ein anderes normales Textprogramm dafür herhalten), Datei anlegen, Text eintippen, Datei speichern, Editor beenden und Compiler aufrufen. Dieser übersetzt das Programm in Maschinen-Quellcode oder einen Zwischencode, zum Beispiel den p-Code von UCSD-Pascal.

Hier scheiden sich dann die (Compiler-)Geister: Manche benötigen einen Assembler, der Maschinencode erzeugt, andere einen Linker, der die Programm-Module zusammenkettet und wieder andere einen Laufzeit-Interpreter, der die Programme – ähnlich einem Basic-Interpreter – in der Zwischensprache ausführt. Wenn der Compiler allerdings einen Fehler meldet, muß der Programmierer wieder mit dem Editor heran und die Fehler ausbessern. Stürzt das Programm gar völlig ab, geht die Fehlersuche ganz von vorne los.

Ganz anders dagegen arbeitet Turbo-Pascal. Der Programmierer gelangt in eine Art »Benutzerebene«, von der aus er Befehle geben kann. So ruft er mit »E« für »Edit« den eingebauten Texteditor auf. Dieser versteht – Segen für alle Wordstar-Freaks – eine Untermenge der Wordstar-Kommandos. Er ist aber auf das Editieren von Pascal-Programmen hin erweitert worden. So gibt es eine »Auto-Indent«-Funktion. Diese bewirkt, daß der Cursor nach Drücken der Enter-Taste nicht an den Anfang der nächsten Bildschirmzeile, sondern unter den Beginn der letzten Programmzeile gesetzt wird. Damit unterstützt Turbo-Pascal wirkungsvoll das Prinzip der Texteinrückungen, die Pascal-Programme so gut lesbar machen. Auch ist der Turbo-Editor ein ganzes Stück schneller als Wordstar. Beispielsweise führt er den Bildschirmaufbau und die Invertierung von Textblöcken mit Höchstgeschwindigkeit durch – auch eine Rechtfertigung für »Turbo« im Namen »Turbo-Pascal«. Dieser Geschwindigkeitsvorteil liegt allerdings zum Teil daran, daß der gesamte bearbeitete Programmtext im Speicher liegen muß, während Wordstar auch Diskettentexte bearbeiten kann. Trotz dieser Einschränkung eignet sich der

Turbo-Editor nebenbei hervorragend zur Textverarbeitung. So sind die Blockkopierbefehle sogar auf dem CPC 464 und CPC 664 von Schneider ohne Speichererweiterung funktionsfähig.

Sobald der Benutzer den Pascal-Quellcode fertig eingegeben hat, drückt er in Wordstar-Manier CTRL-KD und gelangt wieder in die Kommandoebene. Dort kann er das Diskettenlaufwerk wechseln, neue Disketten anmelden, die Datei speichern und andere Dateien laden. »D« gibt ein Directory der angemeldeten Diskette aus und hat gegenüber dem CP/M-Kommando DIR den Vorteil, daß der freie Disketten-Speicherplatz mit angezeigt wird. Der Benutzer erfährt aus dem Hauptmenü auch, wieviel Speicherplatz der Quelltext belegt und wieviel Platz noch frei bleibt. Mit »C« läßt sich der Compiler starten, der das Programm mit extrem hoher Geschwindigkeit (»Turbo«) übersetzt. Er besitzt jede Menge Fehlermeldungen, die meist sehr informativ sind. Sobald er einen Fehler entdeckt, gibt Turbo-Pascal die (meistens) passende Meldung aus und kehrt direkt in den Turbo-Editor zurück. Der Cursor steht dann an der Stelle, an der der Fehler auftrat. Wenn der Speicherplatz knapp wird, können Sie aber auch die Fehlermeldungen aus dem Programm herauswerfen und haben damit etwa 2 KByte mehr Speicherplatz zur Verfügung. Alle Fehler erscheinen dann nur noch mit ihren Nummern, die im – gut gemachten – Handbuch nachzuschlagen sind.

Müssen Sie sich mit fehlendem Speicherplatz herumschlagen (auf dem CPC 464 und CPC 664 verbleiben nur noch etwa 6 beziehungsweise 8 KByte), bleiben aber auch noch andere Wege, große Programme zu übersetzen. Da wären zuerst einmal die Include-Files. Das sind Dateien, die separat auf der Diskette stehen und erst bei der Übersetzung in den Programmcode integriert werden. Außerdem läßt Ihnen Turbo-Pascal die Wahl, ob der Objektcode des Programms im Speicher abgelegt werden soll (Compile To Memory) oder auf Diskette. Bei letzterem können Sie sich zwischen dem Dateityp »COM« und »CHN« entscheiden. Der Unterschied besteht darin, daß im ersten Fall das erzeugte Programm mit einer Laufzeitbibliothek ausgestattet ist. Diese kostet zwar einiges an Speicher- und Diskettenplatz, erzeugt aber Programme, die unabhängig von Turbo-Pascal arbeiten.

Die Programme sind – wie bereits gesagt – reiner Maschinencode. Die 8-Bit-Variante von Turbo-Pascal erzeugt Z80-Code und ist damit eines der wenigen CP/M-Programme, die nicht auf Computern mit 8080/8085-Prozessor gestartet werden können, sondern den Z80-Chip voraussetzen. Andererseits sind die Z80-Erweiterungen sehr leistungsfähig und bewirken kompakteren und schnelleren Maschinencode.

Was den Programmierer aber wohl am brennendsten interessiert, ist der Befehlsvorrat des Compilers – ein weiterer Pluspunkt für Turbo-Pascal. Während viele Pascal-Compiler der niedrigeren Preisklassen »Subsets« des Sprachstandards sind, also nur Teile der Sprache verstehen, ist Turbo-Pascal ein »Superset« – und was für eines!

Niklaus Wirth entwickelte Pascal eigentlich nur als Lehr- und Unterrichtssprache. Dementsprechend fehlen in Standard-Pascal viele nützliche und dringend notwendige Dinge. Turbo-Pascal wartet mit einer ganzen Menge von diesen auf. Da gibt es zum Beispiel den Datentyp STRING, den man nun nicht mehr durch einen ARRAY OF CHAR simulieren muß. Stilgerecht fehlen auch nicht die entsprechenden Funktionen und Prozeduren zur Stringbehandlung. Selbstverständlich wird die Dateibehandlung auf Diskette und Festplatte samt Relativ- und Direktzugriffdateien unterstützt. Neu ist auch der Datentyp BYTE.

Für Kenner ein Plus: Maschinennähe

Für Maschinensprache-Kenner, denen die Entwicklung von reinen Maschinencode-Programmen zu mühsam ist, hält Turbo-Pascal eine Reihe von maschinennahen Erweiterungen bereit: SHR (Shift Right) und SHL (Shift Left) erlauben das Links- und Rechtsschieben von Bits in Zahlen und Variablen. Die Funktionen HI und LO liefern das High- und Lowbyte einer Integervariablen in eine Byte-Variable:

```
Program HiLow;
Var i:Integer;
Begin
 ReadLn(i);
 WriteLn(Hi(i),' ',Lo(i));
End.
```

Dieses kleine Beispielprogramm liest eine 16-Bit-Zahl von der Tastatur ein und gibt deren High- und Lowbyte aus.

Weitere maschinennahe Funktionen sind ADDR zur Ermittlung der Adresse einer Variablen und PORT zum direkten Zugriff auf die Prozessor-Ports. Das Pseudo-Array MEM (»Memory«) simuliert die beliebten PEEK- und POKE-Befehle aus Basic. Diese Sprache erscheint im Gegensatz zu Turbo-Pascal regelrecht archaisch.

Auf einer viel höheren Ebene arbeiten die Befehle zur Verwaltung dynamischer Variablen, nämlich MARK, RELEASE und DISPOSE.

Natürlich präsentiert sich Turbo-Pascal nicht »nackt«. Neben dem TURBO.COM-File, das mit 31 KByte Länge das Hauptprogramm darstellt, sowie TURBO.MSG (System-Meldungen) und TURBO.OVR (ein 2KByte-Overlay) findet sich noch MC.PAS, das als Demonstration der Leistungsfähigkeit des Compilers dient. MC.PAS ist ein kleines Tabellenkalkulationsprogramm, genannt MicroCalc. Es ist verständlicherweise nicht so leistungsfähig wie seine großen Brüder (etwa Multiplan, Visicalc oder Supercalc), aber es erfüllt seinen Zweck recht gut. Es wurde allerdings auch gar nicht zur »richtigen« Arbeit geschrieben, sondern soll vielmehr exemplarisch die Fähigkeiten von Turbo-Pascal zeigen. Wenn die Kalkulationsaufgaben nicht allzu kompliziert werden, erspart es trotzdem das Geld für die Anschaffung eines anderen Programms.

Zum Lieferumfang gehört ferner ein Programm, das andere Pascal-Programme auf dem Drucker auflistet. LISTER.PAS erzeugt an den passenden Stellen einen Seitenvorschub und kann auf Wunsch Zeilennummern in den Text einfügen und Pascal-Schlüsselwörter unterstreichen.

Für die Schneider-Computer gibt es – gegen Aufpreis, versteht sich – eine Grafikerweiterung, die die hervorragenden Fähigkeiten dieser Computer auf diesem Gebiet unterstützt. Unter anderem dient TURTLE.PAS als einfaches Zeichenprogramm nach dem Prinzip der Logo-Schildkröten-Grafiken. WINDOW.PAS zeigt, wie das »Fensterln« unter Pascal funktioniert.

Immer mehr Firmen versuchen, sich an den Erfolg von Turbo-Pascal anzuhängen und liefern Programmsammlungen für alle möglichen Anwendungszwecke. Zum Beispiel werden da Turbo-Lader, Turbo-Machine, Turbo-Screen oder Turbo-Data angeboten. Von Heimsoeth-Software, der deutschen Vertriebsfirma von Turbo-Pascal, stammen unter anderem Turbo-Graphix, Turbo-Editor und Turbo-Gameworks. Doch aufgepaßt: Turbo-Pascal gibt es auch für IBM-Computer unter MS-DOS. Viele der Programmpakete arbeiten nur unter diesem Betriebssystem und sind für CP/M-Benutzer unbrauchbar. Das betrifft zum Beispiel auch einige der gerade genannten Programme.

Leider vernachlässigt Borland zur Zeit etwas die CP/M-80-Version von Turbo-Pascal. So hat die 8-Bit-Variante vom »Lifting« in der neuen 3.0-Version kaum profitiert, während das MS-DOS-Pendant noch weiter verbessert wurde. Dennoch: An Turbo-Pascal gibt es nicht mehr allzu viel besser zu machen. Es ist einfach schon nahezu perfekt.

(Martin Kotulla/hg)

Der Einstieg in Pascal

Sie wollten schon immer wissen, was es mit der vielgerühmten strukturierten Programmierung in Pascal auf sich hat? Oder besitzen Sie gar einen Pascal-Compiler für Ihren Computer? Dann nehmen Sie sich ein wenig Zeit, und studieren Sie folgenden Artikel.

```
PROGRAM SUMME (INPUT, OUTPUT);
(* DAS ERSTE PASCALPROGRAMM ZUR UEBUNG *)
  CONST ANZAHL  = 4;
        SUMME, X : REAL;
        I        : INTEGER;
BEGIN
  WRITELN('Geben Sie bitte ', ANZAHL, ' Zahlen ein!');
  SUMME:= 0.0;
  FOR I:= 1 TO ANZAHL DO
    BEGIN
      READ(X); SUMME:= SUMME + X
    END;
  WRITELN;
  WRITELN('Die Summe betraegt ', SUMME);
  WRITELN('Der Durchschnitt ist ', SUMME / ANZAHL)
END.
```

Listing 1. Dieses Pascal-Programm berechnet Summe und Querschnitt

Zur Eingabe eines Pascal-Programmes, zur Übersetzung und zur Fehlerkorrektur müssen Sie spezielle Hilfsprogramme benutzen, deren Bedienung natürlich an dieser Stelle nicht beschrieben werden kann. In diesen Fällen hilft ein Blick in die Handbücher, die mit jedem Compiler geliefert werden.

Die grundlegenden Eigenschaften der Spache Pascal lassen sich am einfachsten an einem konkreten Beispielprogramm erläutern (Listing 1). Am besten lesen Sie zunächst einmal den gesamten Programmtext durch. Überlegen Sie sich für jede Zeile, welche Bedeutung die Anweisungen haben könnten. Auch ohne Kenntnisse in Pascal werden Sie erkennen, daß dieses Programm vom Benutzer die Eingabe einer Folge von Zahlen erwartet, über die eine Summe gebildet und gedruckt wird.

Auf den ersten Blick fällt bereits die stufenförmige Einrückung der Zeilen auf. Sie ist jedoch neben den Kommentaren zwischen »(*« und »*)« das einzige an einem Pascal-Programm, das keinerlei Bedeutung bei der Übersetzung des Programms besitzt. Genauer gesagt, betrachtet der Compiler ein Pascal-Programm als eine lange Folge von Symbolen (Sonderzeichen und Namen), die beliebig durch Leerzeichen oder Zeilenenden getrennt sein können. Im Prinzip könnte man also ein Pascal-Programm als einen riesigen Bandwurmsatz in eine einzige Zeile schreiben. Andererseits sind die Einrückungen, Leerzeilen und Kommentare die einzige Orientierungshilfe für den menschlichen Leser, um die teilweise komplex geschachtelten Schleifen und Abfragen in einem Programm auf einen Blick zu erkennen, so daß man einen »guten« Programmierstil bereits am systematischen Layout erkennt.

Es folgt also, daß man sich bei der Programmierung nicht auf eine »logische« Einrückung des Textes verlassen darf, sondern vielmehr durch die Verwendung von Sonderzeichen und Interpunktionszeichen (Semikolon, Punkt, Komma) die Interpretation des Programmes steuern muß.

```
PROGRAM Name (INPUT, OUTPUT);

CONST   Konstantendeklarationen

VAR     Variablendeklarationen

PROCEDURE / FUNCTION Unterprogramm

deklarationen

BEGIN
  Anweisung; Anweisung; ...
END.
```

Bild 1. Der prinzipielle Aufbau eines Pascal-Programmes

AND	FILE	NOT	TO
ARRAY	FOR	OF	TYPE
BEGIN	FORWARD	OR	UNTIL
CASE	FUNCTION	PACKED	VAR
CONST	GOTO	PROCEDURE	WHILE
DIV	IF	PROGRAM	WITH
DO	IN	RECORD	
DOWNTO	LABEL	REPEAT	
ELSE	MOD	SET	
END	NIL	THEN	

Tabelle 1. Reservierte Wörter in Pascal

Neben den Sonderzeichen besteht ein Programm aus Wörtern. Dabei unterscheidet man zwischen reservierten Schlüsselwörtern (Wortsymbolen) und frei wählbaren Namen. Schlüsselwörter haben eine feste syntaktische Bedeutung und können nicht als Namen zum Beispiel für Variablen (wie SUMME) verwendet werden. Tabelle 1 zeigt die kurze Liste der reservierten Schlüsselwörter in Pascal. In den folgenden Artikeln werden Sie die Bedeutung der meisten dieser Wörter kennenlernen.

Namen in Pascal bestehen aus einem Buchstaben, dem eine beliebige Reihe von Buchstaben oder Zahlen folgt. Jeder Pascal-Compiler berücksichtigt mindestens die ersten acht Zeichen eines Namens, so daß Sie Namen nicht wie in Basic auf zwei Zeichen Länge verstümmeln müssen. Da in Pascal nicht nur Variable, sondern auch Unterprogramme, Konstanten und Typen mit Namen versehen werden, sollte man bei der Namensgebung möglichst systematisch vorgehen, um aus diesen Namen auf die Bedeutung schließen zu können.

Beispiele für sinnvolle und erlaubte Namen sind: ANFANGSBUCHSTABE, ENDEZEICHEN, GRENZE, FEHLER.

Nachdem Sie jetzt einen Überblick über die Einzelteile (Symbole) eines Pascal-Programms besitzen, wenden wir uns nun dem Zusammenbau dieser Elemente zu einem vollständigen Programm zu. Jedes Programm hat die in Bild 1 gezeigte Struktur:

- Programmkopf
- Deklarationen
- BEGIN
- Anweisungen
- END

Im Rahmen dieses Artikels besteht jeder Programmkopf aus dem Schlüsselwort PROGRAM, einem Namen und der Parameterliste (INPUT,OUTPUT). Die Namen INPUT und OUTPUT zeigen an, daß im Programm Eingaben von der Tastatur und Ausgaben an den Bildschirm vorkommen können.

Der nachfolgende Deklarationsteil definiert alle Namen, die im Programm verwendet werden. Grundsätzlich gilt in Pascal die Regel, daß die Definition eines Namens im Programm-Text vor der Anwendung des Namens in einer anderen Definition oder in einer Anweisung stehen muß. Dadurch können Pascal-Programme in einem einzigen Durchlauf (one pass) compiliert werden, da vor jeder Anwendung eines Namens der Compiler alle Informationen über einen Namen bereits gelesen hat.

In dem Beispielprogramm in Listing 1 wird zum Beispiel eine Konstante ANZAHL mit der Deklaration

```
CONST ANZAHL = 4
```

definiert. Immer wenn im nachfolgenden Programm-Text der Name ANZAHL

auftritt, wird die Konstante 4 compiliert. Außerdem würde der Compiler eine Zuweisung an diese Konstante, beispielsweise mit »ANZAHL := 39«, als Fehler erkennen. Dies ist einer der vielen Vorteile der expliziten Deklaration von Namen: Der Compiler schützt den Programmierer vor der falschen oder doppelten Verwendung eines Namens. Wer sich schon einmal durch ein größeres fremdes Programm gekämpft hat, wird auch wissen, daß die Abfrage

```
IF ANZAHL=MAXIMALANZAHL THEN...
```

verständlicher ist als

```
IF A=49 THEN...
```

Nicht zuletzt erlaubt die Verwendung von Konstanten eine einfache Änderung bestehender Programme. Soll das Programm in Listing 1 später einmal acht Zahlen summieren, so genügt die einmalige Änderung der Konstanten ANZAHL, wodurch sowohl die Obergrenze der FOR-Schleife als auch die Division bei der Durchschnittsbildung angepaßt wird.

Das Programm SUMME enthält noch eine weitere Form von Deklarationen:

```
VAR SUMME, X: REAL;
    I       : INTEGER;
```

Hinter dem Schlüsselwort VAR werden alle Variablen des Programms mit ihrem Typ (zum Beispiel INTEGER) aufgeführt. Der Typ einer Variablen gibt an, welche Werte eine Variable annehmen darf (zum Beispiel Zahlen oder Zeichen). Während in Basic der Typ einer Variablen aus dem Namen hervorgeht (A$, A%, A), wird in Pascal der Name eines Typs bei der Deklaration angegeben. Zunächst wollen wir uns auf die vordefinierten Standard-Typen beschränken:

REAL: Dieser Typ umfaßt die reellen Zahlen (zum Beispiel +1.0, 0.0, -1.2, 2.3E-4).

INTEGER: Werte vom Typ INTEGER sind ganze Zahlen aus einem beschränkten Intervall. Typischerweise sind dies die Zahlen zwischen -32768 bis 32767.

CHAR: Eine Variable vom Typ CHAR kann ein einzelnes Zeichen, wie »A«, »!« oder »4« speichern.

BOOLEAN: Dieser Typ ist Ihnen sicherlich nicht ganz so vertraut wie die übrigen Standard-Typen. Er umfaßt nämlich nur die logischen Werte »wahr« (TRUE) und »falsch« (FALSE). So liefern zum Beispiel Vergleiche ein Ergebnis vom Typ BOOLEAN. Mit booleschen Werten kann man über die logischen Operatoren UND, ODER und NICHT (AND, OR, NOT) »rechnen«, und die Ergebnisse in einer Variablen speichern:

```
27=13          (FALSE)
12>4           (TRUE)
'A'<'B'        (TRUE)
'A'='a'        (FALSE)
```

A	B	A AND B	A OR B	NOT A
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	TRUE	TRUE	TRUE	FALSE

Tabelle 2. Wahrheitstabelle der logischen Operatoren

```
PROGRAM GESCHWINDIGKEITSTEST (INPUT, OUTPUT);
  CONST LETZTERDURCHLAUF = 10000; (* WIEDERHOLE SCHLEIFE SO OFT *)
  VAR X1      : INTEGER;
      X2      : REAL;
      ZAEHLER: INTEGER;
BEGIN
  WRITELN('INTEGER-Schleife gestartet!');
  X1:= 0;
  FOR ZAEHLER:= 1 TO LETZTERDURCHLAUF DO X1:= X1+1;
  WRITELN('INTEGER-Schleife beendet!');
  WRITELN('REAL-Schleife gestartet!');
  X2:= 0;
  FOR ZAEHLER:= 1 TO LETZTERDURCHLAUF DO X2:= X2+1;
  WRITELN('REAL-Schleife beendet!');
END.
```

Listing 2. Ein Geschwindigkeitsvergleich zwischen REAL und INTEGER

```
(A=B) OR (X=Y)
RICHTIG := ERGEBNIS = LOESUNG
FALSCH  := NOT (RICHTIG)
```

Die letzten beiden Zeilen verwenden zwei boolesche Variablen RICHTIG und FALSCH, die mit

```
VAR RICHTIG, FALSCH: BOOLEAN;
```

deklariert werden müssen. Diese Beispiele machen auch den Unterschied zwischen dem Vergleichsoperator »=« und dem Zuweisungsoperator »:=« deutlich. Mit »ERGEBNIS = LOESUNG« wird der Inhalt der Variablen ERGEBNIS mit dem Inhalt der Variablen LOESUNG verglichen. Stimmen beide überein, so liefert der Vergleich das Ergebnis TRUE. Um einer Variablen einen neuen Wert zuzuweisen, verwendet man den Zuweisungsoperator »:=«. Typische Beispiele sind:

```
I:= I+1
ERGEBNIS:= SIN(X)+COS(X)
LOESUNG:= A*B - C*D + 23.0
```

Zur Verdeutlichung der Wirkung der booleschen Operatoren enthält Tabelle 2 eine sogenannte Wahrheitstabelle, die zum Beispiel angibt, daß falsch und wahr gleich falsch ist (FALSE AND TRUE = FALSE).

Warum unterscheidet man aber zwischen den Typen REAL und INTEGER? Für einen Computer ist es wesentlich einfacher, mit ganzen Zahlen zu rechnen oder zu zählen, als Fließkomma-Arithmetik mit reellen Zahlen durchzuführen. Außerdem lassen sich Zahlen vom Typ INTEGER wesentlich kompakter als REAL-Zahlen speichern. Das bedeutet für die Praxis, daß man nur in Ausnahmefällen Variablen vom Typ REAL deklariert, und zwar nur dann, wenn sehr große Zahlen verarbeitet oder Nachkommastellen dargestellt werden müssen. Bei einer genaueren

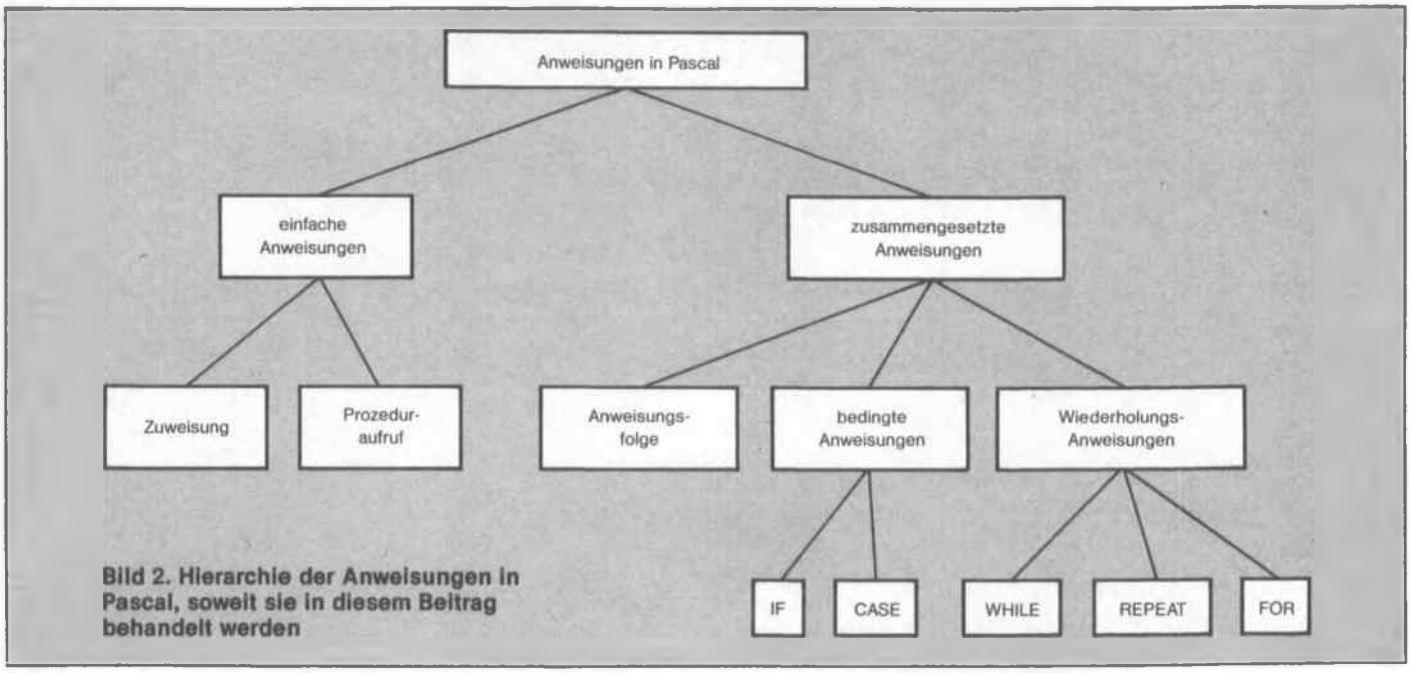


Bild 2. Hierarchie der Anweisungen in Pascal, soweit sie in diesem Beitrag behandelt werden

Untersuchung der meisten Programme wird man jedoch feststellen, daß die Mehrzahl der Variablen nur zum Zählen in Schleifen, als Indizes in Tabellen oder für ähnliche Steuerungsaufgaben benutzt werden, in denen INTEGER-Zahlen völlig ausreichen. Wer einen Computer zur Verfügung hat, sollte sich den erheblichen Geschwindigkeitsunterschied zwischen Operationen mit ganzen und reellen Zahlen am Beispielprogramm in Listing 2 verdeutlichen.

Zum Abschluß dieses kleinen Exkurses über die Standard-Typen in Pascal soll noch ein weiterer Begriff geklärt werden, der Ihnen bei der Lektüre eines Pascal-Lehrbuches oder in Fehlermeldungen des Compilers häufiger begegnen wird: Ein »skalarer Typ« ist ein Typ, dessen Werte einzeln aufzählbar sind und der nicht in einfachere Typen zerlegt werden kann.

INTEGER, CHAR und BOOLEAN sind beispielsweise skalare Typen. Reelle Zahlen oder Mengen haben keinen skalaren Typ. Später werden Anwendungen genannt, in denen nur Werte eines skalaren Typs auftreten dürfen. An solchen Stellen sind also nur ganze Zahlen, nicht aber reelle Zahlen zu verwenden.

```
PROGRAM ZUWEISUNGEN (INPUT, OUTPUT);
  VAR R: REAL;
      I: INTEGER;
      B: BOOLEAN;
      C: CHAR;
BEGIN
  R:= 3.141592653;                  (* PI ist eine reelle Zahl    *)
  R:= SQR(SIN(R)) + SQR(COS(R));    (* -> R = 1.0                 *)
  I:= 0;
  I:= I+1;                          (* vorwaerts zaehlen          *)
  I:= I*I + I - 7;                  (* Ergebnis ist ganze Zahl    *)
  I:= 13 DIV 4;                     (* Division mit Rest -> I=3   *)
  I:= 13 MOD 4;                     (* Divisionsrest -> I=1       *)
  R:= ABS(R);                       (* Absolutwert vom Typ REAL   *)
  I:= ABS(I);                       (* dagl. vom Typ INTEGER      *)
  CH:='A';                          (* nur einzelne Zeichen !     *)
  CH:= CHR(65);                     (* CHR liefert das Zeichen    *)
                                    (* mit dem Code 65, das ist   *)
                                    (* der Buchstabe 'A'          *)
  I:= ORD('A');                     (* ORD wirkt genau umgekehrt, *)
                                    (* also ist danach I= 65      *)
  B:= (I<>4);                       (* TRUE, falls I<>4           *)
  B:= B OR NOT B;                   (* Immer richtig (TRUE)!      *)
END.
```

Listing 3. Mögliche Formen der Zuweisung in Pascal

Anweisungen und Blockstruktur

Jetzt können wir uns dem Anweisungsteil eines Pascal-Programms zuwenden, der definiert, was mit den Objekten geschieht, die im Deklarationsteil vereinbart wurden. Bild 2 zeigt einen Überblick über alle Arten von Anweisungen, die nachfolgend beschrieben sind.

Die Philosophie der sogenannten strukturierten Programmierung, die ja bekanntermaßen der Sprache Pascal zugrunde liegt, verbirgt sich in dem Zweig mit der Bezeichnung »Zusammengesetzte Anweisungen«. Jede zusammengesetzte Anweisung kann aus einer Vielzahl verschiedener (eventuell ebenfalls zusammengesetzter) Anweisungen bestehen. Man kann sich eine Anweisung als einen Block vorstellen, der einen Eingang und einen Ausgang besitzt. Ist ein Block eine zusammengesetzte Anweisung, so enthält er kleinere Blöcke mit je einem Eingang und Ausgang, die nach festen Regeln geschachtelt werden. So kann jeder Block als eine Einheit betrachtet werden, ohne die umgebenden Böcke zu kennen. Das mag etwas abstrakt klingen, aber das Bild einer Blockstruktur ist zur Verdeutlichung der Schachtelungen recht anschaulich.

Einfache Anweisungen: Dies sind die elementaren Bausteine, aus denen der Anweisungsteil besteht. Die wichtigste Form der einfachen Anweisungen hatten wir bereits kennengelernt: Eine Zuweisung besteht aus dem Namen einer Variablen, dem Zuweisungsoperator »:=« und einem Ausdruck. Das Programm in Listing 3 zeigt Ihnen eine Vielzahl von Beispielen für Zuweisungen in Pascal. Sie werden feststellen, daß sich die Ausdrücke von ihren Gegenstücken in Basic oder Fortran nur unerheblich unterscheiden. Sie haben also nicht eine so extravagante Form wie in C oder gar in Forth. Natürlich gilt auch in Pascal die Regel »Punkt- vor Strichrechnung«.

Bitte achten Sie darauf, daß der Typ des Ausdruckes auf der rechten Seite des »:=« mit dem Typ der Variablen vor dem »:=« verträglich ist. Daß man einer Variablen vom Typ CHAR keine Zahl zuweisen kann, ist wohl selbstverständlich. Jedoch ist es auch nicht möglich, einer Variablen vom Typ INTEGER direkt eine reelle Zahl zuzuweisen. Man

Fortsetzung auf Seite 30

Fortsetzung von Seite 28

muß vielmehr angeben, was mit den eventuell vorhandenen Nachkommastellen geschieht. Hierzu dienen die Standardfunktionen TRUNC und ROUND, die Nachkommastellen einer reellen Zahl abschneiden oder zur nächsten ganzen Zahl aufrunden (Tabelle 3).

X =	TRUNC(X)	ROUND(X)
1.234	1	1
1.765	1	2
-1.234	-1	-1
-1.77	-1	-2

Tabelle 3. Die Wirkung von TRUNC und ROUND

Wie bestimmt man aber den Typ eines Ausdrucks? Dazu stellt man zunächst die Typen der Konstanten und Variablen (aus dem Deklarationsteil) fest. Eine Zahl ohne Dezimalpunkt und Exponent ist vom Typ INTEGER:
Beispiele:

```
1  0  -8 100000
```

(Zahlen vom Typ INTEGER)

```
1.2   1E-4    123.4
```

(Zahlen vom Typ REAL)

Wird eine reelle Zahl mit einer ganzen Zahl verknüpft, so erhält man als Ergebnis eine reelle Zahl. Indem man schrittweise (unter Beachtung der Prioritäten und Klammern) den Ausdruck auflöst, entsteht der Ergebnistyp.
Beispiele:

```
A:= 1000 * (1000 + 4)
B:= 1000 * (1000 + 4.0)
```

Da der Wertebereich INTEGER meist nur Zahlen kleiner oder gleich 32768 darstellen kann, würde bei der Ausführung der ersten Zuweisung ein Überlauf eintreten. Durch die Addition von 4.0 (vom Typ REAL) ist jedoch im zweiten Fall das Ergebnis (B) vom Typ REAL und somit noch darstellbar.

Tabelle 4 gibt einen Überlick über die möglichen Operatoren in Pascal mit konkreten Beispielen. Außergewöhnlich ist nur die Tatsache, daß die logischen Operatoren AND und OR stärker binden als die Vergleichsoperatoren (>,<,=,<>). Diese Einheit muß bei der Formulierung von Bedingungen (zum Beispiel bei IF-Anweisungen) durch eine korrekte Klammerung beachtet werden:
Beispiel:

```
IF (A=B) OR (C=D) THEN...
```

und nicht

```
IF A=B OR C=D THEN...
```

(Beim zweiten Ausdruck würde zunächst »B OR C« berechnet, das Ergebnis mit A verglichen, das Resultat (vom Typ BOOLEAN) schließlich mit D verglichen werden, was wohl nicht beabsichtigt war.)

Operator	Operation mit Beispiel	Operandentypen	Ergebnistyp
+ (Vorz.)	Identitaet + 3.4 (= 3.4)	INTEGER, REAL	wie Operand
- (Vorz.)	Vorzeichenwechsel - 3.4 (=-3.4)	INTEGER, REAL	wie Operand
+	Addition 1 + 1 (=2)	INTEGER, REAL	wie Operand
	Vereinigungsmenge [1,2,3,4] + [3,5] (= [1,2,3,5])	Mengen	Menge
-	Subtraktion 1 - 1 (= 0)	INTEGER, REAL	wie Operand
	Differenzmenge [1,2,3,4] - [3,5] (=[1,2,4])	Mengen	Menge
*	Multiplikation 1 * 1 (=1)	INTEGER, REAL	wie Operand
	Schnittmenge [1,2,3,4] * [3,5] (= [3])	Mengen	Menge
DIV	Division mit Rest 7 DIV 3 (=2)	INTEGER	INTEGER
MOD	Divisionsrest 7 DIV 3 (=1)	INTEGER	INTEGER
/	'normale' Division 1 / 4 (=0.25)	INTEGER, REAL	REAL (immer)
=	gleich 'A'='a' [1,2,3]=[] 'Otto' = 'Anna (alle FALSE)	Skalar, Pointer Menge, String	BOOLEAN
<>	ungleich (analog)	Skalar, Pointer	BOOLEAN
<	kleiner 'A'<'B' (= TRUE)	Skalar, String	BOOLEAN
>	groesser 'Otto'>'Anna' (=TRUE)	Skalar, String	BOOLEAN
>=	groesser oder gleich Test auf Obermenge [1,2,3]>=[1] (= TRUE)	Skalar, String Menge	BOOLEAN BOOLEAN
<=	kleiner oder gleich Test auf Teilmenge [1,2,3,5] <= [1..4] (=FALSE)	Skalar, String Menge	BOOLEAN BOOLEAN
IN	Test auf Enthaltensein 33 IN [1..40] (=TRUE)	Skalar und Menge	BOOLEAN
NOT	nicht NOT FALSE (=TRUE)	BOOLEAN	BOOLEAN
AND	und (I>J) AND (J>I) (=FALSE)	BOOLEAN	BOOLEAN
OR	oder (A=B) OR (A<>B) (=TRUE)	BOOLEAN	BOOLEAN

Tabelle 4. Operatoren und Ihre Wirkungen

Erwähnenswert ist die Tatsache, daß es in Standard-Pascal keinen Exponentialoperator (A hoch B) gibt. Der weiterführende Artikel über Prozeduren und Funktionen beschreibt, wie man sich diesen Operator selbst definiert.

Um die Ergebnisse von Berechnungen und die Inhalte von Variablen am Bildschirm darzustellen und auch um Eingaben des Benutzers von der Tastatur zu lesen, gibt es in Pascal einige vordefinierte Unterprogramme. Diese Prozeduren werden durch die Angabe ihres Namens, den »Prozeduraufruf« aktiviert. Das Programm aus Listing 1 enthält bereits verschiedene Prozeduraufrufe. Mit WRITELN (write line) wird zum Beispiel der Cursor am Bildschirm auf den Anfang der nächsten Zeile gesetzt. Oft muß man einer Prozedur weitere Informationen übergeben. Diese »Parameter« werden in Klammern und durch Kommata getrennt hinter dem Prozedurnamen aufgeführt:

```
WRITE('SUMME = ', A+B+C);
WRITE(' DM')
```

Als Parameter für die Prozedur WRITE sind neben Textkonstanten (Strings in Hochkommata) alle Werte der Standard-Typen (REAL, INTEGER, CHAR und auch BOOLEAN) zugelassen. Gerade für Einsteiger bietet sich damit die Chance, während des Programmablaufs Zwischenergebnisse oder Variablen ohne großen Aufwand anzuzeigen. Zwei aufeinanderfolgende WRITE-Anweisungen drucken ihre Werte ohne Zwischenraum direkt hintereinander. Möchte man nach der Ausgabe einen Zeilenvorschub durchführen, so verwendet man WRITELN statt WRITE. Das Gegenstück zu WRITE und WRITELN sind die Prozeduren READ und READLN. Als Parameter an diese Prozeduren übergibt man Variablen, denen Werte zugewiesen sind, die von der Tastatur eingelesen werden. Betrachten wir folgendes Beispiel:

```
READ(A,B,C,D)
```

Dieser Prozeduraufruf liest vier Werte ein. Ob es sich dabei um ganzzahlige oder reelle Zahlen oder um Zeichen handelt, hängt von der Deklaration der Variablen A, B, C und D ab. Sind A, B, C und D Variablen vom Typ INTEGER, so kann der Benutzer folgende Eingaben machen:

```
1 2 3 4 [RETURN Taste]
```

oder auch in mehreren Zeilen:

```
1 2 [RETURN Taste]
3   [RETURN Taste]
4   [RETURN Taste]
```

Mit »READLN (A); READLN(B); READLN(C); READLN(D)« wird nach dem Einlesen jedes Wertes der Rest der Eingabezeile ignoriert, so daß die vier Werte in vier Zeilen eingegeben werden müssen.

Später werden Sie sehen, wie man eigene Prozeduren und Funktionen in Pascal definiert, denen man wie den Standard-Funktionen Parameter übergeben kann, oder die Ergebnisse zurückliefern. Solche benutzerdefinierten Prozeduren entsprechen also im Prinzip den Unterprogrammen (Stichwort GOSUB, RETURN) in Basic.

Jetzt sind alle einfachen Anweisungen bekannt, so daß wir mit der Besprechung der zusammengesetzten Anweisungen fortfahren können.

```
BEGIN (* TAUSCHE A <-> B *)
  H:= A;
  A:= B;
  B:= H
END
```

Listing 4. Ein Beispiel für einen abgeschlossenen Anweisungsblock

Eine »Anweisungsfolge« faßt eine Reihe von Anweisungen zu einer einzigen Anweisung zusammen. Listing 4 zeigt als Beispiel für die allgemeine Struktur einer Anweisungsfolge ein kurzes Programmstück, in dem der Inhalt der Variablen A und B vertauscht wird. Dieses Diagramm soll zeigen, wie die kleinen Blöcke (Anweisungen) zu einem großen Block (der zusammengesetzten Anweisung) zusammengefaßt werden. Bitte beachten Sie, daß Semikola zwischen den Anweisungen stehen und nicht hinter jeder Anweisung. Daher muß vor dem abschließenden END kein Semikolon stehen. Jedoch ist es erlaubt, überflüssige Zeichen in eine Anweisungsfolge einzufügen, so daß folgende Anweisungsfolgen ebenfalls syntaktisch korrekt sind:

```
BEGIN H:=A; A:=B; B:=H; END
```

und

```
BEGIN ;H:=A;; A:=B; B:=H END
```

Der Anweisungsteil eines Pascal-Programms ist also syntaktisch gesehen eine Anweisungsfolge. Da die Anweisungen in einer Anweisungsfolge immer in derselben Reihenfolge abgearbeitet werden, benötigt man zur bedingten Ausführung von Befehlen eine zusätzliche zusammengesetzte Anweisung.

Bedingungen auswerten mit IF

In Pascal gibt es zwei Formen der »IF-Anweisung«, die in Listing 5 und Listing 6 dargestellt sind. Die Bilder 3 und 4 sind die zu diesen Listings passenden Struktogramme. Die Bedingung nach dem Schlüsselwort IF ist ein beliebiger Ausdruck mit dem Ergebnistyp BOOLEAN. Ist der Wert des Ausdruckes TRUE, so wird die Anweisung hinter dem Schlüsselwort THEN ausgeführt. Ist das Ergebnis FALSE, so wird die Anweisung nach dem Schlüsselwort ELSE (falls vorhanden) ausgeführt.

Soll hinter THEN oder ELSE mehr als eine einzelne Anweisung stehen, müssen Sie diese Anweisungen mit BEGIN und END zu einer zusammengesetzten Anweisung »klammern«. Ein Beispiel hierfür zeigt Listing 7. Ein häufiger Fehler besteht darin, vor ELSE ein Semikolon einzufügen. In diesem Fall erkennt der Compiler die einseitige Auswahl

```
IF Bedingung THEN
  Anweisung
```

```
IF A=X THEN
  WRITELN(X, 'gefunden!')
```

Listing 5. Die einfache IF-Anweisung

```
IF Bedingung THEN
  Anweisung
ELSE
  Anweisung
```

```
IF A>B THEN
    MAXIMUM:= A
ELSE
    MAXIMUM:= B
```

Listing 6. Die IF-Anweisung mit ELSE-Zweig

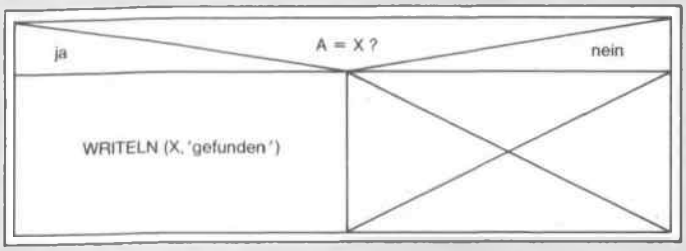


Bild 3. Struktogramm der einfachen IF-Anweisung

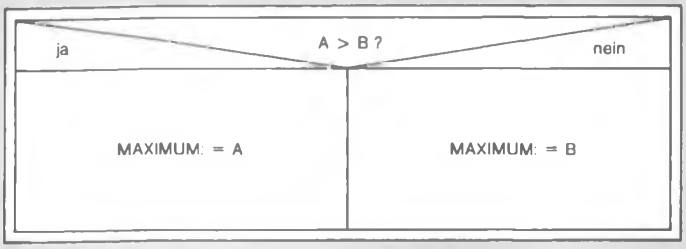


Bild 4. Struktogramm der IF-Anweisung mit ELSE-Zweig

```
PROGRAM PQFORMEL (INPUT, OUTPUT);
(* BERECHNE DIE LOESUNG DER GLEICHUNG X*X + P*X + Q = 0 *)
  VAR  P,Q,A,W:  REAL;
BEGIN
  WRITE('P = '); READLN(P);
  WRITE('Q = '); READLN(Q);
  A:= - P / 2;
  W:= SQR(A) - Q;
  IF W<0 THEN
    WRITE('Es existiert keine Loesung')
  ELSE
    BEGIN
      WRITELN('X1 = ', A + SQRT(W));
      WRITELN('X2 = ', A - SQRT(W))
    END;
END.
```

Listing 7. »Quadratische Gleichung« als Beispiel für die IF-THEN-ELSE-Anweisung

```
PROGRAM MAXIMUM (INPUT, OUTPUT);
  VAR MAX  : INTEGER;
      A,B,C: INTEGER;
BEGIN
  WRITE('A B C:'); READLN(A,B,C);
  IF A>B THEN
    IF A>C THEN
      MAX:= A
    ELSE
      MAX:= C
  ELSE
    IF B>C THEN
      MAX:= B
    ELSE
      MAX:= C;          (*<- erst hier darf ein Semikolon stehen! *)
  WRITELN('Das Maximum von ', A, B, C,' ist ',MAX)
END.
```

Listing 8. »Maximum« – ein Beispiel für verschachtelte IF-Abfragen

(wie in Bild 3) und »beschwert« sich, daß er mit dem Schlüsselwort ELSE nichts anzufangen weiß.

Natürlich kann die Anweisung nach THEN und ELSE wiederum eine IF-Anweisung sein, was in Listing 8 dazu verwendet wird, das Maximum der Zahlen A, B und C zu bestimmen. Bild 5 zeigt die zugehörige Blockstruktur, wobei man deutlich erkennt, daß jeder der geschachtelten Blöcke nur einen Eingang und einen Ausgang besitzt. Ein Beispiel für die Schachtelung der einseitigen Auswahl bietet Listing 9. Hier wird zu einem Tagesdatum das Datum des nachfolgenden Tages berechnet, wobei zur Vereinfachung angenommen wird, daß jeder Monat 30 Tage besitzt.

Wollen Sie komplexe IF..THEN..ELSE-Schachtelungen programmieren, so müssen Sie darauf achten, daß der Compiler jedes ELSE der letzten IF-

Anweisung zuordnet, die noch kein ELSE besitzt. Um in diesen Fällen Probleme zu vermeiden, empfiehlt es sich, die »innen« liegenden IF-Anweisungen mit BEGIN und END zu klammern.

Oft ist es erforderlich, in Abhängigkeit eines Wertes verschiedene Berechnungen durchzuführen. Für solche Fälle bietet sich die »CASE-Anweisung« an (Listing 10 und Bild 6). Es wird der Ausdruck nach dem Schlüsselwort CASE berechnet (der einen skalaren Typ besitzen muß) und in Abhängigkeit vom Ergebnis zu einer der Anweisungen hinter den Fallmarken verzweigt. Diese Fallmarken müssen jeweils Konstanten mit dem Typ des Ausdruckes hinter CASE sein. Stimmt keine der Fallmarken mit dem Ergebnis überein (zum Beispiel MONAT = 14), so erfolgt in Standard-Pascal ein Programmabbruch mit Fehlermeldung (viele Compiler erlauben jedoch auch die Angabe eines ELSE-Zweiges, auch OTHERWISE genannt, dessen Anweisung in diesem Fall ausgeführt wird).

Ein weiteres Beispiel für die CASE-Anweisung ist in Listing 11 gegeben. In diesem Programm werden von der Tastatur zwei Zahlen gelesen, die in Abhängigkeit von einem Zeichen (* . + - /) multipliziert, addiert, subtrahiert oder dividiert werden. Falls bei der Verarbeitung kein Fehler aufgetreten ist, wird das Ergebnis ausgedruckt. Bemerkenswert an dem Beispiel ist noch die Verwendung der booleschen Variablen OK, die den Wert FALSE enthält, falls ein illegaler Befehl eingegeben oder eine Division durch 0 versucht wurde.

In Basic würde man bei einem Fehler mit GOTO aus dem Inneren der CASE-Anweisung springen. Die auf den ersten Blick etwas schwerfällig anmutende Lösung mit der booleschen Variablen sorgt jedoch dafür, daß die CASE-Anweisung keinen zusätzlichen »Fehler-Ausgang« besitzt, der die Blockstruktur der zusammengesetzten Anweisung verletzen würde. Gerade bei großen Programmen sind nämlich Sonderbehandlungen mit Sprüngen quer durch den gesamten Programmtext eine schwer kontrollierbare Fehlerquelle.

Als kleiner Einschub ist in Listing 12 eine alternative Lösung der Aufgabe in dem Programm in Listing 10 gegeben. Sie nutzt die Tatsache, daß man in Pascal auch mit Mengen wie in der Mengenlehre in der Grundschule rechnen kann. Eckige Klammern ersetzen hierbei die geschweiften Klammern der Mathematik.

[1,2,3,4] und [1..4] bezeichnen dieselbe Menge, nämlich die Zahlen von 1 bis 4. Hingegen bezeichnen [] aber auch [3..1] die leere Menge, die keine Zahl enthält.

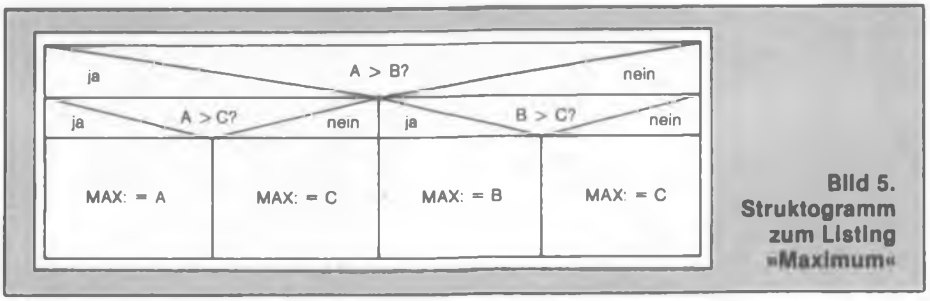


Bild 5. Struktogramm zum Listing »Maximum«

```
PROGRAM TAGESDATUM (INPUT, OUTPUT);
  CONST TAGE_PRO_MONAT  = 30;
        MONATE_PRO_JAHR = 12;
  VAR TAG, MONAT, JAHR: INTEGER;
BEGIN
  ...
  WRITELN('Heute ist der ', TAG, '.', MONAT, '.', JAHR);
  TAG:= TAG+1;
  IF TAG>TAGE_PRO_MONAT THEN
    BEGIN
      TAG:=1; MONAT:= MONAT+1;
      IF MONAT>MONATE_PRO_JAHR THEN
        BEGIN
          MONAT:=1; JAHR:= JAHR+1
        END
    END;
  WRITELN('Morgen ist der ', TAG, '.', MONAT, '.', JAHR);
END.
```

Listing 9. »Tagesdatum« – ein Beispiel zur Schachtelung der einseitigen Auswahl

```
CASE Ausdruck OF
  Fallmarke, ..., Fallmarke : Anweisung;
  Fallmarke, ..., Fallmarke : Anweisung;
  ...
  Fallmarke, ..., Fallmarke : Anweisung
END


CASE MONAT OF
  1,3,5,7,8,10,12: TAGE_PRO_MONAT:= 31;
  4,6,9,11       : TAGE_PRO_MONAT:= 30;
  2              : BEGIN
                     SCHALTJAHR:= (JAHR MOD 4) = 0) AND
                                  ((JAHR MOD 100)<>0) OR
                                   (JAHR MOD 400)=0);
                     IF SCHALTJAHR THEN TAGE_PRO_MONAT:= 29
                                   ELSE TAGE_PRO_MONAT:= 28
                   END;
END; (* CASE*)
```

Listing 10. Allgemeine Form und Beispiel zur CASE-Anweisung

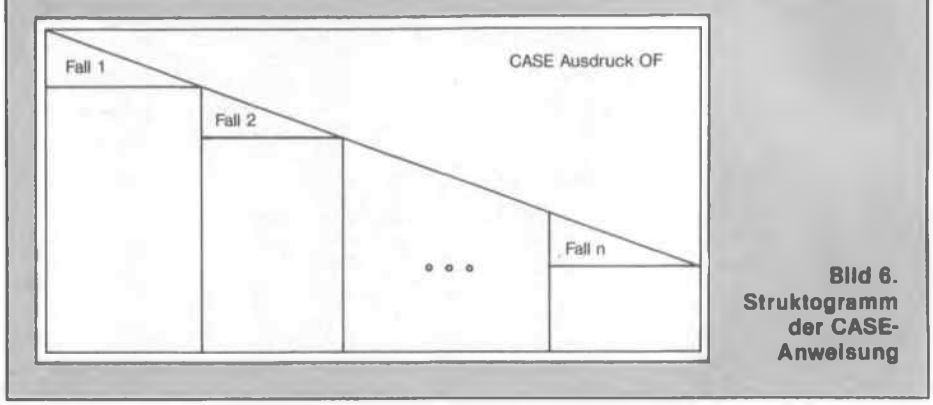


Bild 6. Struktogramm der CASE-Anweisung

Mit `X IN [1,3,9,27]` wird geprüft, ob die Zahl X in der Menge der Dreierpotenzen bis 27 enthalten ist. In der Tabelle 4 auf Seite 30 sind einige dieser Operationen mit Mengen dargestellt (Vereinigungsmenge, Schnittmenge, Differenzmenge, Test auf Teilmengen, Test auf Gleichheit).

Bei der Besprechung der zusammengesetzten Anweisungen haben wir in der Übersicht (Bild 2) die »Wiederholungsanweisungen« erreicht. Es gibt in Pascal drei verschiedene zusammengesetzte Anweisungen, die Wiederholungen definieren, von denen sich jede für spezielle Anwendungsfälle besonders eignet.

Möchte man eine Anweisung so oft ausführen, bis eine gewisse Bedingung erfüllt wird, bietet sich die »REPEAT-Anweisung« (Listing 13) an. Zwischen den Schlüsselwörtern REPEAT und UNTIL steht eine beliebige Folge von Anweisungen, die Semikola trennen. Am Ende der Ausführung dieser Anweisungen wird der Ausdruck hinter dem

```
PROGRAM MINICOMPUTER (INPUT, OUTPUT);
  VAR CH      : CHAR;
      A,B, ERG: REAL;
      OK      : BOOLEAN;
BEGIN
  WRITELN('Geben Sie ein Operationszeichen (+,-,*,.,/) und zwei Zahlen ein!');
  READLN(CH, A, B);
  OK:= TRUE;                (* noch ist kein Fehler aufgetreten *)
  CASE CH OF
    '.','*' : ERG:= A * B;
    '/'     : IF B = 0 THEN
                BEGIN OK:= FALSE;
                  WRITELN('Fehler: Division durch Null!')
                END
              ELSE
                ERG:= A / B;
    '+'     : ERG:= A + B;
    '-'     : ERG:= A - B;
    ELSE      BEGIN
                OK:= FALSE;
                 WRITELN('Die Operation ', CH, ' ist nicht moeglich!')
              END;
  END; (* von CASE *)
  IF OK THEN
     WRITELN('Ergebnis: ', ERG)
END.
```

Listing 11. Ein »Minicomputer« als weiteres Beispiel für CASE

Listing 12. Das Beispiel aus Listing 9, programmiert unter Verwendung von Mengen

```
IF MONAT IN [1,3,5,7,8,10,12] THEN TAGE_PRO_MONAT:= 31 ELSE
IF MONAT IN [4,6,9,11]        THEN TAGE_PRO_MONAT:= 30 ELSE
ELSE
  BEGIN (* hier ist MONAT = 2 *)
    SCHALTJAHR:= (JAHR MOD 4) = 0) AND
                 ((JAHR MOD 100)<>0) OR
                  (JAHR MOD 400)=0);
    IF SCHALTJAHR THEN TAGE_PRO_MONAT:= 29
    ELSE TAGE_PRO_MONAT:= 28
  END;
```

```
REPEAT
  Anweisung;
  Anweisung;
  ...
  Anweisung
UNTIL Bedingung
```

```
REPEAT
  WRITELN('Alles klar? (Ja oder Nein)');
  READ(CH)
UNTIL CH IN ['J','j','N','n']
```

Listing 13. Allgemeine Form und Beispiel für REPEAT...UNTIL

```
PROGRAMM WURZEL (INPUT, OUTPUT);
  CONST EPS = 1.0E-7;      (* Genauigkeit: mindestens 7 Nachkommastellen*)
  VAR   X,Y,Z: REAL;
BEGIN
  WRITE('Die Quadratwurzel aus '); READ (X);
  IF X<0 THEN
    WRITELN(' ist keine reelle Zahl')
  ELSE
    BEGIN   (* 1. einen Startwert Z berechnen *)
      Y:= 2;
      REPEAT
        Z:= Y; Y:= Y*Y
      UNTIL Y>X;

      (* 2. Jetzt folgt die eigentliche Berechnung: *)
      REPEAT
        Y:= Z;
        Z:= 0.5 * (Y + X/Y)
       UNTIL ABS(Y-Z)<= EPS;      (* bis Abweichung kleiner als Genauigkeit *)

      WRITELN(' ist ', Z)
    END;
END.
```

Listing 14. Berechnung von Quadratwurzeln unter Verwendung von REPEAT...UNTIL-Schleifen

Schlüsselwort UNTIL ausgewertet. Er liefert einen booleschen Wert (TRUE oder FALSE). Ist die Abbruchbedingung nicht erfüllt, so wird die Anweisungsfolge wiederholt. In dem Beispiel werden also so lange Zeichen von der Tastatur eingelesen, bis der Benutzer ein »J« oder ein »N« eingegeben hat. Bei näherem Hinsehen können Sie also feststellen, daß Mengenoperationen auch auf Mengen von Zeichen anwendbar sind:

```
CH IN ['J', 'J', 'n', 'N']
```

Eine REPEAT-Schleife benutzt man bei Wiederholungen, die in der Abbruchbedingung einen Wert benötigen, der erst im Inneren der Schleife ermittelt wird. Als Beispiel ist in Listing 14 die Berechnung der Quadratwurzel für eine reelle Zahl X durch eine Iteration aufgeführt. Dieses Programm berechnet die Wurzel in zwei Schritten, die jeweils eine REPEAT-Schleife benötigen. Im ersten Schritt wird die kleinste Zweierpotenz (2,4,8,16,...) Z berechnet, die quadriert gerade größer als X ist. Anschließend wird in der zweiten REPEAT-Schleife dieser Startwert Z schrittweise modifiziert, bis die Differenz zwischen zwei Iterationswerten kleiner als die gewünschte Genauigkeit EPS ist.

Viel häufiger als die REPEAT-Anweisung findet die WHILE-Anweisung Verwendung (Listing 15 und Bild 8). Hier wird die Bedingung nach dem Wortsymbol WHILE vor der Ausführung der Anweisung nach DO geprüft. Damit besteht insbesondere die Möglichkeit, daß diese Anweisung kein einziges Mal ausgeführt wird, falls nämlich die Bedingung bereits beim Eintritt in die Schleife den Wert FALSE liefert.

Soll die WHILE-Schleife, was wohl in der Mehrzahl der Fälle zutreffen wird, mehrere Anweisungen umfassen, so muß man diese zu einer Anweisungsfolge mit BEGIN und END klammern. Ein kleines, aber besonders schönes Beispiel für die Funktionsweise der WHILE-Schleife zeigt Listing 16. Hier wird der Wert $E = N^K$ für natürliche Zahlen N und K berechnet. K gibt also an, wie oft N mit sich selbst multipliziert werden muß, um E zu erhalten. Daher wird im Programm die Zählvariable I verwendet, die von K abwärts gegen 0 zählt. Das Beispiel ist deshalb so ideal, da durch die Eigenschaft der WHILE-Anweisung, eine Prüfung am Beginn der Schleife durchzuführen, auch die Sonderfälle korrekt behandelt werden. In der Mathematik gilt nämlich:

$N^0 = 1$ für alle N

$0^K = 0$ für alle $K \neq 0$

Durch eine geeignete Wahl der Wiederholungsanweisung kann man sich also viele Sonderbehandlungen mit IF-Anweisungen ersparen.

Zum Vergleich ist in Listing 17 je eine WHILE- und eine REPEAT-Schleife angegeben, die alle Zahlen zwischen A und B druckt. Wenn Sie mit diesem Programm experimentieren, werden Sie feststellen, daß der einzige Unterschied bei der Ausführung darin besteht, daß in der REPEAT-Schleife für A > B die Zahl A gedruckt wird, während die WHILE-Schleife in diesem Fall nicht in Funktion tritt.

Für eine Anwendung wie in Listing 17 wird man jedoch normalerweise die dritte Variante der Wiederholungsanweisungen in Pascal benutzen: Die FOR-Schleife (Listing 18) findet überall dort Anwendung, wo die Anzahl der Schleifendurchläufe vor dem Beginn der Schleife bereits bekannt ist. Den genauen Ablauf der Ausführung soll fol-

gendes Beispiel zeigen:

```
FOR I:= A TO B DO WRITE(I)
```

1. Berechne A und weise den Wert der Variablen I zu.
2. Berechne B und speichere den Wert in einer temporären (unsichtbaren) Variablen X
3. Ist I>X, so beende die Schleife
4. Sonst drucke die Zahl I
5. Erhöhe I um 1 und weiter bei 3

Die Variable I muß einen skalaren Typ besitzen (also zum Beispiel CHAR, aber nicht REAL) und mit dem Ergebnistyp von A und B verträglich sein. Aus dem obigen Schema (1..6) folgt außerdem, daß für A>B wie bei der WHILE-Schleife keine Zahl gedruckt wird. Man könnte in der Schleife die Variable B auch beliebig verändern, ohne die obere Grenze der Zählvariablen (die ja in einer temporären Variablen gespeichert wurde) zu beeinflussen.

Um rückwärts zu zählen, besteht noch die Möglichkeit, eine Variante der FOR-Schleife zu verwenden:

```
FOR I:= A DOWNTO B DO WRITE(I)
```

Die obigen Regeln gelten für diese Schleife analog. Mit der FOR-Schleife kann man nur in Einerschritten aufwärts oder abwärts zählen, andere Schrittweiten müssen mit einer WHILE-Schleife explizit programmiert werden. Im Beispiel aus Listing 19 wird ein Winkel PHI in Schritten von DELTA hochgezählt und dabei eine Funktion am Bildschirm »gezeichnet«.

Wenn Sie zu der Übersicht in Bild 2 zurückblättern, werden Sie erkennen, daß wir mit der Vorstellung der FOR-Schleife praktisch alle Anweisungen in Pascal besprochen haben. Zwei fundamentale Eigenschaften von Pascal wurden jedoch noch ausgespart.

(1) Zusammengesetzte Typen ermöglichen es, nicht nur mit einzelnen Zahlen und Zeichen, sondern sogar mit hierarchisch aufgebauten und auch veränderlichen Datenstrukturen zu arbeiten.

(2) Neben den bereits vorgestellten Standard-Prozeduren READ, READLN, WRITE und WRITELN gibt es noch eine Vielzahl an vordefinierten Prozeduren und Funktionen, die in jeder Implementation der Sprache Pascal vorhanden sind. Viel bemerkenswerter ist jedoch die Möglichkeit, beliebige Teile eines Programmes als Prozeduren und Funktionen zu definieren, die über bestimmte Schnittstellen mit ihrer »Umwelt« (Prozeduren oder Hauptprogramm) kommunizieren können.

In den folgenden Beiträgen werden diese beiden wichtigen und interessanten Gebiete ausführlich behandelt, wobei gleichzeitig das bisherige Wissen über die einfachen Typen und die Anweisungen in Pascal angewandt und vertieft wird. (Florian Matthes/ev)

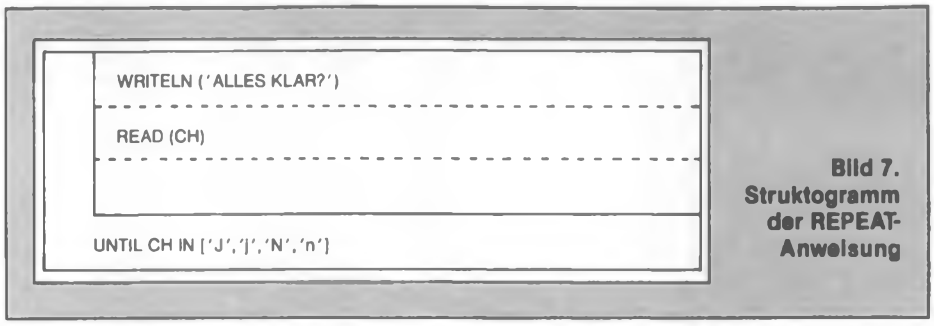


Bild 7. Struktogramm der REPEAT-Anweisung

```
WHILE Bedingung DO
  Anweisung

WHILE X>=1.0 DO
  X:= X / 2.0;
```

Listing 15. Allgemeine Form und Beispiel für die WHILE...DO-Schleife

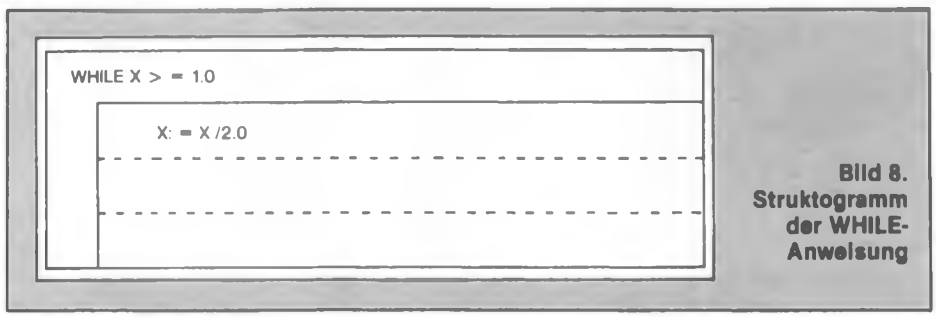


Bild 8. Struktogramm der WHILE-Anweisung

```
PROGRAM HOCH (INPUT, OUTPUT);
  VAR I,N,E,K: INTEGER;
BEGIN
  READLN(N,K);
  IF (N<0) OR (K<0) THEN
    WRITELN('ungueltige Eingabe!')
  ELSE
    BEGIN
      E:=1; I:=K;
      WHILE I>0 DO
        BEGIN
          E:= E*N;
          I:= I-1
        END;
      WRITELN(N, '^', K, '=', E)
    END
END.
```

Listing 16. Ein Demo-Programm unter Verwendung der WHILE...DO-Schleife

```
PROGRAM ZUM_VERGLEICH (INPUT, OUTPUT);
  VAR I, A, B: INTEGER;
BEGIN
  READLN(A,B);
  WRITE('Mit REPEAT:');
  I:=A;
  REPEAT
    WRITE(I);
    I:= I+1
  UNTIL I>B;
  WRITELN;

  WRITE('Mit WHILE:');
  I:= A;
  WHILE I<=B DO
    BEGIN
      WRITE(I);
      I:= I+1
    END;
  WRITELN
END.
```

Listing 17. Ein Programm zum Vergleich zwischen WHILE und REPEAT

```
FOR Variable:= Ausdruck TO Ausdruck DO
  Anweisung

FOR Variable:= Ausdruck DOWNTO Ausdruck DO
  Anweisung

FOR I:= 1 TO 1000 DO
  SUMME:= SUMME + 1 / I

FOR I:= 1000 DOWNTO 1 DO
  SUMME:= SUMME + 1 / I
```

Listing 18. Allgemeine Form und Beispiel für die FOR-Schleife

```
PROGRAM SCHWINGUNG (INPUT, OUTPUT);
(* ZEICHE DIE FUNKTION F(PHI) = EXP(-PHI/PI) * SIN (PHI) *)
  CONST PI = 3.1415926;
        MAXZEILEN = 40;          (* Laenge der Ausgabe in Zeilen       *)
        MITTE = 40;              (* Mitte eines 80-Zeichen Bildschirms*)
  VAR   PHI,Y, DELTA : REAL;
        SPALTE, ZEILE: INTEGER;

BEGIN
  DELTA:= 6*PI / MAXZEILEN
  FOR ZEILE:= 0 TO MAXZEILEN DO
    BEGIN
      PHI:= DELTA * ZEILE;     (* AKTUELLER WINKEL *)
      Y  := EXP(-PHI/PI)*SIN(PHI);
      SPALTE:= MITTE + ROUND(MITTE * Y);

      WRITELN('*': SPALTE);    (* drucke Stern in Spalte SPALTE *)
    END;
END.
```

Listing 19. Schwingungs-Berechnung mit FOR-Schleife

Programmieren mit Pascal

Mit diesem Artikel steigen wir voll in die eigentliche Programmierung mit Pascal ein. Prozeduren, Funktionen und Datentypen heißen die Stichworte.

Wie steigt man am einfachsten in die Programmierung ein? Nun, vermutlich mit einem Programm zum Ausprobieren und Experimentieren. Das Beispielprogramm in Listing 1 liest einen Text von der Tastatur ein und zeichnet ein einfaches Balkendiagramm wie in Bild 1, das die Häufigkeit jedes Zeichens angibt. Wir möchten Sie bitten, vor dem Weiterlesen einen längeren Blick auf das Listing zu werfen, um einerseits die bereits im einführenden Artikel besprochenen Sprachelemente (zum Beispiel die FOR-Schleife) zu verstehen und andererseits einen Überblick über die neuen Probleme zu bekommen.

Der Kern des Programms besteht darin, für jeden Buchstaben einen Zähler zu führen, der bei jedem Auftreten des Buchstabens im eingelesenen Text erhöht wird. Natürlich möchte man nicht für jeden Buchstaben eine Zählvariable deklarieren und dann in einer riesigen CASE-Anweisung jeden einzelnen Zähler erhöhen. Statt dessen vereinbart man ein ARRAY (Feld, Tabelle) von Zählern:

```
VAR Z: ARRAY ['0'..'Z']
OF INTEGER;
```

Ein Array ist eine Variable, über deren Namen mehrere Werte des gleichen Typs angesprochen werden. Das Array Z besitzt für jedes Zeichen zwischen »0« und »Z« einen Wert vom Typ INTEGER. Auf die einzelnen Werte greift man durch Nennung des Array-Namens, gefolgt von einem Index in eckigen Klammern, zu:

```
Z['0']:= 13;
WRITELN (Z['X']);
Z[CH]:= Z[CH] + 1
```

Die letzte Zuweisung zeigt eine wichtige Eigenschaft von Arrays: Man kann als Index eine Variable (hier CH) oder einen beliebig komplexen Ausdruck verwenden. Für Basic-Programmierer ungewohnt ist die Eigenschaft von Pascal, nicht nur Zahlen, sondern beliebige geforderte Datentypen als Index zuzulassen.

Um alle Zähler in Z zurückzusetzen, kann man also folgende FOR-Schleife verwenden:

```
FOR CH:= '0' TO 'Z' DO Z[CH]:= 0;
```

Nach diesem speziellen Beispiel sollen Sie auch die allgemeinen Regeln für Arrays in Pascal kennenlernen:

1. Ein Array muß wie jede andere Variable im Variablendeklarationsteil vereinbart werden.
2. Ein Array ist ein zusammengesetzter Typ, der durch zwei Typen beschrieben wird: Den skalaren Indextyp und den beliebig wählbaren Elementtyp. Z hat also den Indextyp »'0' .. 'Z'« und den Elementtyp INTEGER.
3. Beim Indizieren muß der Typ des

```
PROGRAM HAEUFIGKEIT (INPUT, OUTPUT);
(* STATISTIK DER BUCHSTABENHAEUFIGKEIT IN EINEM TEXT *)

  CONST VON = '0'; BIS = 'Z';
        ENDEZEICHEN = '@';
        BILDSCHIRMBREITE = 80;          (* 80 ZEICHEN PRO ZEILE AM BILDSCHIRM *)

  TYPE  ZAEHLERFELD = ARRAY[VON..BIS] OF INTEGER;

  VAR   Z : ZAEHLERFELD;

  (* ------------------------------------------------- BEGINN DER PROZEDUREN *)

  PROCEDURE LOESCHEN(VAR A: ZAEHLERFELD);
  (* ALLE ZAEHLER ZURUECKSETZEN *)

    VAR CH: CHAR;

  BEGIN
    FOR CH:= VON TO BIS DO A[CH]:=0
  END; (* LOESCHEN *)

  PROCEDURE ZAEHLEN (VAR A: ZAEHLERFELD);
  (* TEXT VON DER TASTATUR BIS ENDEZEICHEN EINLESEN UND ZAEHLER IN A ERHOEHEN *)

    VAR CH: CHAR;

  BEGIN
    WRITELN('GEBEN SIE JETZT BITTE EINEN TEXT EIN, DER MIT ', ENDEZEICHEN,
            ' ENDET');
    REPEAT
      READ(CH);
      IF CH IN [VON..BIS] THEN
        A[CH]:= A[CH] + 1;
    UNTIL CH = ENDEZEICHEN;
    WRITELN
  END; (* ZAEHLEN *)

  PROCEDURE STATISTIK (A: ZAEHLERFELD);
  (* DRUCKE FUER JEDEN BUCHSTABEN EINEN BALKEN, DER DIE HAEUFIGKEIT DES      *)
  (* ZEICHENS ANGIBT.                                                        *)

    VAR CH    : CHAR;
        MAX   : INTEGER;
        HOEHE: INTEGER;

      PROCEDURE BALKEN (N:INTEGER);
      (* ZEICHNE EINEN HORIZONTALEN BALKEN MIT N ZEICHEN *)

        BEGIN
          WHILE N>0 DO
            BEGIN
              WRITE('#'); N:= N-1
            END;
          WRITELN;
        END; (* BALKEN *)

  BEGIN (* HIER BEGINNT DER ANWEISUNGSTEIL DER PROZEDUR STATISTIK *)
    WRITELN('STATISTIK DER BUCHSTABENHAEUFIGKEITEN:');

    (* BESTIMME ZUNAECHST DEN MAXIMALEN ZAEHLERSTAND: *)
    MAX:= A[VON];
    FOR CH:= SUCC(VON) TO BIS DO
      IF A[CH]>MAX THEN MAX:= A[CH];
    (* ZEICHEN JETZT DIE BALKEN: *)
    FOR CH:=VON TO BIS DO
      BEGIN
        WRITE(CH, ' ');
        HOEHE:= ROUND( A[CH] * (BILDSCHIRMBREITE-3)/MAX);
        (* DAMIT NIMMT DER HOECHSTE BALKEN DIE GESAMTE BILDSCHIRMBREITE EIN   *)
        BALKEN (HOEHE);
      END;
  END; (* STATISTIK *)

  (* ------------------------------------------------- ENDE DER PROZEDUREN *)

BEGIN
  LOESCHEN (Z);
  ZAEHLEN  (Z);
  STATISTIK(Z);
END.
```

Listing 1. Programm »Häufigkeit«

Ausdruckes in eckigen Klammern mit dem bei der Deklaration vereinbarten Indextyp verträglich sein. »Z[5]:=0« wäre nach der oben angegebenen Deklaration von Z nicht zulässig, da die Zahl 5 kein Zeichen ist. Außerdem ist auch die Zuweisung »Z['!']:=3« nicht erlaubt, da das Zeichen '!' nicht im Intervall »0« bis »Z« liegt.

Wenn Sie die Deklaration der Variablen Z im Beispiel 24 untersuchen, werden Sie feststellen, daß diese in drei Schritten erfolgte: Zunächst wurden zwei Konstanten

```
CONST VON='0'; BIS='Z';
```

eingeführt. Mit diesen beiden Konstanten erfolgte im nächsten Schritt eine Typendeklaration:

```
TYPE ZAEHLERFELD = ARRAY
[VON .. BIS] OF INTEGER;
```

Nach dem Schlüsselwort TYPE wird ein Name genannt, dem nach einem Gleichheitszeichen eine Typangabe folgt. Im dritten Schritt kann in der Variablendeklaration der Typbezeichner ZAEHLERFELD benutzt werden, um ein Array mit den Indexgrenzen VON und BIS und dem Elementtyp INTEGER zu definieren.

```
VAR A,B,FELD: ZAEHLERFELD;
    ZAEHLER: ZAEHLERFELD;
```

Neben der Einsparung von Schreibarbeit erhöhen Typnamen die Lesbarkeit von Programmen. Listing 2 zeigt einige Typdeklarationen und anschließend die Anwendung dieser Typnamen in weiteren Typ- und Variablendeklarationen.

Jetzt werden Sie wahrscheinlich alle Anweisungen mit den Array-Variablen A und Z verstehen. Neben der Verwendung von Arrays und Typnamen enthält das Programm »Häufigkeit« (Listing 1) noch ein weiteres neues Sprachelement von Pascal. Sie finden in dem Programm viermal das Schlüsselwort PROCEDURE, das eine Prozedurdeklaration einleitet. Nach diesem Schlüsselwort folgt der Name der Prozedur, über den die Prozedur aufgerufen wird. Der eigentliche Anweisungsteil des Programmes (nach dem letzten BEGIN) besteht also nur aus den folgenden drei Prozeduraufrufen:

```
LOESCHEN (Z);
ZAEHLEN  (Z);
STATISTIK(Z);
```

Das Programm besteht also aus drei voneinander unabhängigen Programmteilen zum Rücksetzen der Zähler, zum Einlesen des Textes und der Ausgabe der Statistik. Jeder Prozedur wird in Klammern das Array Z als aktueller Parameter übergeben.

Wie sieht aber eine Prozedurdeklaration aus? Prozedurdeklarationen müssen am Ende des Vereinbarungsteils (also hinter den Konstanten, Typ- und Variablendeklarationen) stehen. Die

```
 0 ¦ ##################################################
 1 ¦ ##########################################
 2 ¦ ##################################
 3 ¦ ##########################
 4 ¦ ##########################
 5 ¦
 6 ¦
 7 ¦ ############################################################
 8 ¦ ######################################
 9 ¦ ########################################################################
 : ¦
 ; ¦
```

Bild 1. Ergebnisausdruck des Programms »Häufigkeit« (Ausschnitt)

Prozedurdeklaration selbst besteht aus drei Teilen:

1. Der Prozedurkopf enthält nach dem Namen der Prozdur bei Bedarf in Klammern eine Liste der formalen Parameter:

```
PROCEDURE LOESCHE
(VAR A: ZAEHLFELD);
```

Die Prozedur LOESCHEN muß also mit einer Variablen vom Typ ZAEHLERFELD aufgerufen werden. Die Variable besitzt innerhalb der Prozedur den Namen A. Beinhaltet eine Prozedur mehrere Parameter, so müssen die aktuellen Parameter in der Reihenfolge der formalen Parameter angegeben werden. Der Aufbau der Parameterliste kommt später genauer zur Sprache.

2. Nach dem Prozedurkopf folgen alle Deklarationen, wie sie auch beim Hauptprogramm möglich sind. Das heißt, daß eine Prozedur wie ein eigenständiges Programm mit eigenen Konstanten, Typen, Variablen und wiederum Prozeduren ausgestattet sein kann. So besitzt zum Beispiel die Prozedur LOESCHEN eine Variable CH vom Typ CHAR, die nur innerhalb der Prozedur gültig ist, und damit insbesondere nicht von Anweisungen im Hauptprogramm oder in anderen Prozeduren verändert werden kann. Die Prozedur STATISTIK enthält selbst eine Prozedurdeklaration (BALKEN).

3. Schließlich folgt zwischen den Schlüsselworten BEGIN und END der Anweisungsteil der Prozedur. Es ist üblich, nach dem letzten END in Kommentarklammern den Namen der Prozedur zu nennen, die an dieser Stelle endet, damit man bei großen Programmen jeden Anweisungsteil leicht dem Prozedurkopf zuordnen kann.

Im Beispielprogramm »Häufigkeit« werden also nacheinander die Anweisungsteile der Prozeduren LOESCHEN, LESEN und STATISTIK ausgeführt, wodurch nacheinander die Zähler Z gelöscht, erhöht und schließlich grafisch dargestellt werden.

Jetzt ist es an der Zeit, daß Sie etwas mehr über die Sichtbarkeitsregeln von Namen in Pascal erfahren. Sie legen fest, welche Namen in einer Prozedur

```
PROGRAM TYPEN (INPUT, OUTPUT);

  CONST ANZAHL_AUFGABEN = 400;

  TYPE GROSSE_ZAHL = REAL;
       KLEINE_ZAHL = INTEGER;
       ZEICHEN     = CHAR;
       ERGEBNIS    = BOOLEAN;
       EINKOMMEN   = ARRAY [1970..1986]
       OF GROSSE_ZAHL;
       ZEILE       = ARRAY [1..80] OF
       CHAR;
       TESTBOGEN   = ARRAY [1..ANZAHL_
       AUFGABEN] OF ERGEBNIS;

VAR  A,B,C                    : GROSSE_ZAHL;
     CH1,CH2                  : ZEICHEN;
     UEBERSCHRIFT             : ZEILE;
     DEUTSCHLAND, ENGLAND     : EINKOMMEN;
     MEDIZINERTEST            : TESTBOGEN;

BEGIN
END.
```

Listing 2. Beispiele für Typdeklarationen

zur Anwendung kommen. Zunächst noch eine Definition: Ein Block ist eine Prozedur oder das Hauptprogramm selbst.

In einem Block sind alle die Namen sichtbar (also gültig), die innerhalb dieses Blockes deklariert wurden. Außerdem noch diejenigen, die in einem »umfassenden« Block deklariert wurden. Wurde ein Name sowohl in einem umgebenden Block als auch im Block selbst deklariert, so ist nur die innere Deklaration sichtbar.

Variablen, die in einer Prozedur festgelegt wurden, heißen lokale Variablen dieser Prozedur. Benutzt jedoch eine Prozedur Variablen, die in einem umgebenden Block deklariert wurden, so nennt man diese global.

Diese Regeln soll das Programm »Sichtbarkeit« in Listing 3 verdeutlichen. Zur Unterstützung ist in Bild 2 die Schachtelung der Blöcke grafisch dargestellt. Die Prozedur P1 besitzt drei lokale Namen, nämlich den Parameter V1, die Konstante K4 und die Variable V 2. Durch die Deklaration des Namens »V1:CHAR« in P1 ist der Name »V1: T1« aus dem Hauptprogramm in P1 nicht sichtbar. Andererseits sind die Namen von P1 nicht in P2, P21 oder dem Hauptprogramm sichtbar.

Prozedur P21 zeigt auch, daß Namen über mehrere Blöcke hinweg sichtbar

sind: So in P21 der Name K2 aus dem Hauptprogramm, da P21 in der Prozedur P2 enthalten ist, die selbst zum Hauptprogramm gehört.

Für die Praxis bedeutet dies, daß man in einer Prozedur Namen ohne Rücksicht auf die »Umgebung« deklarieren kann. Tatsächlich versucht man sogar, nur in Ausnahmefällen auf globale Variablen zuzugreifen und alle Werte als Parameter an die Prozeduren zu übergeben. Damit kann man später die Prozedur in einem völlig anderen Programm verwenden, ohne daß Namenskonflikte auftreten.

Nach diesem etwas theoretischen Abschnitt soll eine Reihe von Beispielen diese Regeln illustrieren und außerdem verschiedene Typen von Parametern vorstellen. Das erste Beispiel (Listing 4) zeigt eine Prozedur, die eine INTEGER-Zahl als eine Hexadezimal-Zahl anzeigt. Zur Anzeige wird die vierstellige Zahl zunächst in zwei Byte zerlegt, deren beide Hexadezimal-Stellen schließlich mit der Prozedur PRINTDIGIT angezeigt werden. Klar zu erkennen ist hier die Schachtelung der drei Prozeduren, wobei jede einen Parameter X vom Typ INTEGER besitzt.

Die Prozedur SWAP in Listing 5 besitzt zwei Parameter des Typs INTEGER. Die Prozedur tauscht den Inhalt dieser beiden Parameter aus.

Listing 4 und 5 zeigen die beiden in Pascal vorhandenen Typen von Parametern. Wird ein formaler Parameter im Prozedurkopf mit dem Schlüsselwort VAR gekennzeichnet, so nennt man ihn einen Variablenparameter. In diesem Fall muß der aktuelle Parameter eine Variable des angegebenen Typs sein. Der Aufruf

```
SWAP(5,6)
```

wäre also nicht zulässig. Innerhalb der Prozedur bewirkt jede Zuweisung an einen Variablenparameter eine Änderung des Wertes der Variablen, die als aktueller Parameter übergeben wurde. Wurde also zum Beispiel die Prozedur SWAP mit den Variablen X und Y aufgerufen, so wird durch die Zuweisung A:=B tatsächlich der Variablen X der Wert der Variablen Y zugewiesen.

Die Parameter der Prozedur PRINTHEX sind hingegen Wertparameter. Beim Aufruf einer solchen Prozedur werden die aktuellen Parameter, die nicht unbedingt Variablen sein müssen, in lokalen Variablen der Prozedur gespeichert. Bei der Ausführung der Prozedur wird nur auf diese Kopie des aktuellen Parameters zugegriffen, so daß insbesondere eine Zuweisung an einen Wertparameter niemals eine Änderung im aufrufenden Block bewirkt.

Ist Ihnen der Unterschied noch nicht völlig klar, so sollten Sie das Schlüsselwort VAR aus dem Prozedurkopf von SWAP löschen und das Programm neu compilieren. Nach der Rückkehr aus der Prozedur werden dann die Werte der Variablen X und Y im Hauptprogramm unverändert sein.

Jeder Parameter wird wie eine Variable mit der Angabe seines Typs nach einem Doppelpunkt deklariert. Dabei ist

```
PROGRAM VARIABLENPARAMETER (INPUT,
  OUTPUT);
  VAR X,Y: INTEGER;

  PROCEDURE SWAP (VAR A,B: INTEGER);
  (* TAUSCHE DEN INHALT VON A UND B *)

    VAR H: INTEGER;

  BEGIN
    H:=A; A:=B; B:=H
  END; (* SWAP *)

BEGIN
  X:= 3; Y:= 4;
  SWAP (X,Y);
  WRITELN (X,Y);
END
```

Listing 5. Ein Beispiel für die Verwendung von Wert-Parametern

```
PROGRAM SICHTBARKEIT (OUTPUT);

  CONST K1 = 3; K2 = 'C'; K3 = '----------';
  TYPE  T1 = INTEGER; T2 = REAL;
  VAR   V1 : T1;

  PROCEDURE P1 (VAR V1:CHAR);

    CONST K4 = 23.4;
    VAR   V2 : INTEGER;

  BEGIN
    (* HIER SIND FOLGENDE NAMEN SICHTBAR:
      CONST K1 = 3; K2 = 'C'; K3 = '----------'; K4= 23.4;
      TYPE  T1 = INTEGER; T2 = REAL;
      VAR   V1 : CHAR;
            V2 : INTEGER;          *)
  END; (* P1 *)

  PROCEDURE P2;
    VAR V2: REAL;

    PROCEDURE P21;
      CONST K1 = '2';

    BEGIN
    (* HIER SIND FOLGENDE NAMEN SICHTBAR:
      CONST K1 = '2'; K2 = 'C'; K3 = '----------';
      TYPE  T1 = INTEGER; T2 = REAL;
      VAR   V1 : T1;
            V2 : REAL;          *)
    END; (* P21 *)
  BEGIN (* ANFANG DES ANWEISUNGSTEILS VON P1 *)
    (* HIER SIND FOLGENDE NAMEN SICHTBAR:
      CONST K1 = 3; K2 = 'C'; K3 = '----------';
      TYPE  T1 = INTEGER; T2 = REAL;
      VAR   V1 : T1;
            V2 : REAL;          *)
  END; (* P2 *)

BEGIN
  (* HIER SIND FOLGENDE NAMEN SICHTBAR:
    CONST K1 = 3; K2 = 'C'; K3 = '----------';
    TYPE  T1 = INTEGER; T2 = REAL;
    VAR   V1 : T1;
          V2 : REAL;          *)
END.
```

Listing 3. Ein Beispiel für Sichtbarkeitsregeln

```
PROGRAM WERTPARAMETER (INPUT, OUTPUT);

  VAR X: INTEGER;

  PROCEDURE PRINTHEX (X: INTEGER);
  (* DRUCKE GANZE ZAHL X ALS HEXADEZIMALZAHL IM FORMAT
  $XXXX *)
  PROCEDURE PRINTBYTE (X: INTEGER);
  (* DRUCKE ZAHL ZWISCHEN 0 UND 255 ALS BYTE IM FORMAT
  XX *)

      PRINTDIGIT (X: INTEGER);
      (* DRUCKE HEXADEZIMALE ZIFFER *)

      BEGIN
        IF X>9 THEN
          WRITE(CHR(X-10+ORD('A')))
        ELSE
          WRITE(X)
      END; (* PRINTDIGIT *)

    BEGIN
      PRINTDIGIT(X DIV 16);
      PRINTDIGIT(X MOD 16);
    END; (* PRINTBYTE *)

  BEGIN
    WRITE('$');
    PRINTBYTE(X DIV 256);
    PRINTBYTE(X MOD 256);
  END; (* PRINTHEX *)

BEGIN (* HAUPTPROGRAMM *)
  WRITELN('Geben Sie positive ganze Zahlen ein:');
  READ(X);
  WHILE X>0 DO
    BEGIN
      WRITELN(' = '); PRINTHEX(X);
      READ(X)
    END;
END.
```

Listing 4. Umrechnung dezimal nach hexadezimal

```
PROGRAM VEKTOROPERATIONEN (INPUT, OUTPUT);

  CONST N = 3; (* LAENGE EINES VEKTORS = ANZAHL DER ELEMENTE
  IM VEKTOR *)

  TYPE  VEKTOR = ARRAY[1..3] OF REAL;

  VAR   X,Y,Z : VEKTOR;
        SKALAR: INTEGER;

  PROCEDURE HOLEN( VAR V: VEKTOR);
  (* VEKTOR MIT N KOMPONENTEN EINLESEN *)

    VAR I: INTEGER;

  BEGIN
    FOR I:= 1 TO N DO READ(V[I]);
    READLN;
  END; (* HOLEN *)

  PROCEDURE DRUCKEN( V: VEKTOR);
  (* VEKTOR MIT N KOMPONENTEN DRUCKEN *)

    VAR I: INTEGER;

  BEGIN
    FOR I:= 1 TO N DO WRITE(V[I]:8);
    WRITELN;
  END; (* DRUCKEN *)

  PROCEDURE ADDIEREN (A,B: VEKTOR; VAR C: VEKTOR);
  (* C = A + B. DIE ADDITION ERFOLGT KOMPONENTENWEISE *)

    VAR I: INTEGER;

  BEGIN
    FOR I:= 1 TO N DO C[I]:= A[I] + B[I]
  END; (* ADDIEREN *)


  PROCEDURE MULTIPLIZIEREN (S: REAL; A: VEKTOR; VAR C: VEKTOR)
  (* C = S * A. JEDE KOMPONENTE WIRD MIT S MULTIPLIZIERT *)

    VAR I: INTEGER;

  BEGIN
    FOR I:= 1 TO N DO C[I]:= S * A[I]
  END; (* MULTIPLIZIEREN *)

BEGIN
  WRITE('X ='); HOLEN(X);
  WRITE('Y ='); HOLEN(Y);
  ADDIEREN(X,Y,Z);

  WRITE('X + Y ='); DRUCKEN(Z);
  WRITE('F ='); READLN(SKALAR);
  MULTIPLIZIEREN (SKALAR,X,Z);
  WRITE('F * X ='); DRUCKEN(Z);
END.
```

Listing 6. Vektoroperationen in Pascal

zu beachten, daß in der Parameterliste nur Typnamen auftreten dürfen. Diese müssen Sie also eventuell zunächst (außerhalb der Prozedur) im Typvereinbarungsteil bestimmt haben. Statt

```
PROCEDURE DRUCKE(X:ARRAY [1..2]
OF CHAR);
```

muß man schreiben

```
TYPE T: ARRAY [1..2] OF CHAR;
PROCEDURE DRUCKE(X: T);
```

Es gibt keinerlei Beschränkung für die Typen der Variablen- oder Wertparameter. Dies soll das Beispiel in Listing 6 verdeutlichen. In der Mathematik würde man ein Array mit N Elementen des Typs REAL als einen Vektor reeller Zahlen bezeichnen. Mit Vektoren kann man wie mit »normalen« Zahlen rechnen. Sind X und Y zwei Vektoren und S eine reelle Zahl, so kann man zum Beispiel X+Y und S*X berechnen. Damit Sie auch mit diesen Operationen ein wenig experimentieren können, sind neben den Prozeduren ADDIEREN und MULTIPLIZIEREN noch die Prozeduren HOLEN und DRUCKEN zur Ein- und Ausgabe von Vektoren vorhanden.

An diesen vier Prozeduren erkennen Sie gut den Unterschied zwischen Wert- und Variablenparametern. Nur wenn über einen Parameter ein Ergebnis oder eine Eingabe zurückgeliefert werden soll, verwendet man Variablenparameter, ansonsten Wertparameter. Dadurch ist während der Ausführung der aktuelle Parameter gegen (unbeabsichtigtes) Überschreiben geschützt.

Es gibt jedoch einen weiteren Fall, in dem man mit Variablenparametern arbeitet, obwohl keine Ergebnisse zurückgeliefert werden sollen. Werden sehr große Variablen an ein Unterprogramm übergeben, so existiert jeder Parameter im Speicher des Rechners doppelt. Einerseits wird der Wert der Variablen im aufrufenden Programm gespeichert und zusätzlich beim Aufruf der Prozedur als lokale Variable. Für die Praxis können Sie sich also merken, daß Sie große Arrays (insbesondere auf Mikrocomputern mit ihrem kleinen adressierbaren Speicherraum) am besten als Variablenparameter übergeben. Bei diesen wird nämlich nur eine Adresse (also nur wenige Bytes) an die Prozedur übergeben, die die Position des aktuellen Parameters im Speicher bezeichnet. In der Prozedur wird dann jeder Zugriff auf diesen Parameter indirekt über die Adresse ausgeführt.

Wegen dieser unterschiedlichen Formen der Übergabe bezeichnet man den Aufruf mit Wertparametern als »call by value« und den Aufruf mit Variablenparametern als »call by reference«.

Zu diesem Themengebiet der »Technik hinter den Kulissen« gehört auch die Verwaltung des Speichers bei einem Pascal-Rechner. Neben den (statistischen) Sichtbarkeitsregeln für die Namen von Variablen, muß auch die (dynamische) Gültigkeit der Werte von Variablen Beachtung finden.

Beim Eintritt in einen Block ist der Wert jeder Variablen, die nicht Parameter einer Prozedur oder Funktion ist, undefiniert.

So ist zum Beispiel am Programmanfang jede Variable unbestimmt. Sie besitzt also nicht etwa wie in Basic den Wert Null. Bei jedem neuen Aufruf einer Prozedur verhält es sich mit allen lokalen Variablen bis auf die Parameter ebenso. Man kann also nicht davon ausgehen, daß die Variablen die Werte ihres letzten Aufrufes beibehalten.

Vielleicht interessiert es Sie, die Hintergründe dieser Regel zu erfahren. Bei der Übersetzung eines Pascal-Programms bestimmt der Compiler für jede Prozedur den Speicherplatzbedarf für alle lokalen Variablen. Innerhalb dieses Speichers weist er jeder Variablen eine feste Position zu. Jedoch bleibt die absolute Lage des Speicherblockes im Computer unbestimmt.

Erst während der Ausführung wird beim Aufruf jeder Prozedur der Speicherblock für die lokalen Variablen reserviert. Umgekehrt wird beim Ende der Ausführung einer Prozedur ihr gesamter Speicherblock wieder freigegeben. Unterprogramme haben aber bekanntlich die Eigenschaft, das zuletzt aufgerufene Unterprogramm als erstes wieder zu verlassen. Ein Beispiel:

```
A ruft B
  B ruft C
    C ruft D
      D kehrt zurück zu C
    C kehrt zurück zu B
  B kehrt zurück zu A
Ende von A
```

Deshalb werden die Variablen eines Pascal-Programms auf einem Stack (Stapelspeicher) verwaltet.

Betrachten wir das Programm in Listing 7. Es besteht aus zwei Prozeduren, die in verschiedener Reihenfolge aufgerufen werden (ansonsten aber nicht viel Sinnvolles erledigen). Wir wollen nun nach jedem Schritt einen Blick auf den Speicher des Rechners werfen (Bild 3):

Zu Programmbeginn (1) ist der gesamte Speicher frei. Beim Eintritt in das Hauptprogramm wird zunächst Platz für die Variable M des Hauptprogramms geschaffen (2). Nun erfolgt der Aufruf der Prozedur P1, die ihrerseits Platz für die Variable L1 benötigt (3). Nach der Rückkehr aus P1 kann dieser Platz sofort wieder freigegeben werden (4). Der Aufruf P2(FALSE) erfolgt in denselben Schritten (5 und 6), wobei jedoch Platz für zwei Variablen (L2 und auch P1) benötigt wird.

Sie sehen schon jetzt, daß derselbe Speicherplatz sowohl für die Variablen von P1 als auch von P2 verwendet wird. Beim Aufruf von P2(TRUE) (7), ergibt sich zunächst der Zustand von (5), jedoch wird außerdem in P2 noch P1 aufgerufen, so daß sich schließlich eine Speicherverteilung wie in (8) ergibt. Offensichtlich liegt beim zweiten Aufruf von P1 die Variable L2 an einer anderen absoluten Adresse. Bei der Rückkehr aus P1(9) und P2(10) werden wieder die lokalen Variablenbereiche freigegeben.

Der Stack »wächst« also von unten nach oben und nimmt von dort wieder nach unten ab, wobei er immer einen zusammenhängenden Speicherbereich bildet. Daß Ihr Rechner zur Laufzeit der Programme einen Stack verwaltet, merken Sie spätestens dann, falls bei einem Prozeduraufruf kein Platz mehr für die lokalen Variablen vorhanden ist. Das quittiert das Programm gewöhnlich mit der Fehlermeldung »stack overflow«.

Funktionen

Inzwischen sind Sie mehrmals auf die Formulierung »Prozedur« oder »Funktion« gestoßen, ohne daß Sie Näheres über Funktionen in Pascal erfahren haben. Eine Funktion ist eine spezielle Form einer Prozedur, die zusätzlich noch einen Wert als Ergebnis liefert. In der Mathematik gibt es zum Beispiel die Maximumfunktion, die das Maximum von zwei Zahlen liefert, so daß gilt:

```
max {3,4} = 4
3 + max{0,-7} = 3
```

Man kann also das Ergebnis der Funktion auch in arithmetischen Ausdrücken verwenden. All diese Möglichkeiten gelten bei der Verwendung von Funktionen in Pascal, die einige Standardfunktionen der Mathematik nachbilden. Dabei verbirgt sich unter der Funktion HOCH (Listing 8) die in Pascal standardmäßig nicht vorhandene Methode, um A hoch B für beliebige Zahlen zu berechnen.

Bei der Definition unterscheidet sich eine Funktion von einer Prozedur nur durch den Funktionskopf. Hier ersetzt das Schlüsselwort FUNCTION das Wortsymbol PROCEDURE. Außerdem wird zusätzlich am Ende der (eventuell leeren) Parameterliste nach einem Doppelpunkt der Name des Typs angegeben, zu dem das Ergebnis der Funktion gehört. Somit lautet der Funktionskopf der Funktion MAX, die das (reelle) Maximum zweier reeller Zahlen berechnet, folgendermaßen:

```
FUNCTION MAX (A,B: REAL): REAL;
```

Der Ergebnistyp darf kein zusammengesetzer Typ, wie zum Beispiel eine Menge oder ein Array, sein. Um innerhalb der Funktion das Ergebnis zu bestimmen, verwendet man den Funktionsnamen in einer Zuweisung:

```
IF A>B THEN
   MAX:=A
ELSE
MAX:=B
```

Natürlich muß auf jeden Fall innerhalb einer Funktion eine solche Zuweisung stattfinden, damit die Funktion bei der Rückkehr einen definierten Wert liefert. Funktionsaufrufe sind nur innerhalb von Ausdrücken zulässig, während Prozeduraufrufe syntaktisch gesehen Anweisungen sind.

```
IF MAX(A,B) >4 THEN ...
A := MAX(A,B);
MIN:= -MAX(-A,-B)
```

Jetzt folgt eine kleine Sammlung von Prozeduren und Funktionen, die Ihnen die Programmierung typischer Operationen mit Arrays zeigt. Die erste Prozedur in Listing 9 sortiert den Inhalt des Arrays in A, das als Parameter übergeben wird. Durch die Wahl von SORTTYPE = INTEGER könnten Sie diese Prozedur auch zum Sortieren ganzer Zahlen (oder jedes anderen Typs) verwenden. Das Sortierverfahren ist eines der einfachsten und langsamsten überhaupt. Bei jedem Durchlauf der äußeren FOR-Schleife wird jeweils ein Wert im Array an seine korrekte Position gebracht und dazu in der inneren Schleife der maximale Wert im Restarray bestimmt. Somit wird im ersten Durchlauf die größte Zahl mit der Zahl an der letzten Arrayposition vertauscht. Im nächsten Durchlauf wechselt die zweitgrößte Zahl mit der vorletzten Arrayposition den Platz, bis im letzten Durchlauf der kleinste Wert an der ersten Position landet.

Eine etwas exotischere Prozedur stellt Listing 10 vor. Hier wird der Kehrwert des ganzzahligen Parameters I exakt gedruckt. Bei periodischen Brüchen erscheint ein Pfeil am Beginn der Periode.

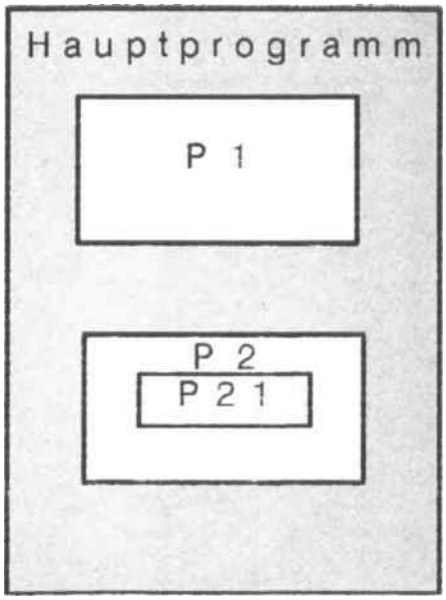


Bild 2. Statische Schachtelung von Prozeduren

```
PROGRAM SPEICHERPLATZ (INPUT, OUTPUT);

  VAR M: REAL;

  PROCEDURE P1;

    VAR L1: INTEGER;

  BEGIN
    WRITELN ('PROZEDUR P1 MIT L1 =',
  L1);
END; (* P1 *)


PROCEDURE P2 (AUCHP1: BOOLEAN);

  VAR L2: REAL;

BEGIN
  WRITELN('PROZEDUR P2 MIT L2 =',
    L2);
    IF AUCHP1 THEN P1;
  END; (* P2 *)


BEGIN
  P1;
  P2 (FALSE);
  P2 (TRUE)
END.
```

Listing 7. Beispielprogramm zur Speicherorganisation (vergleiche Bild 3).

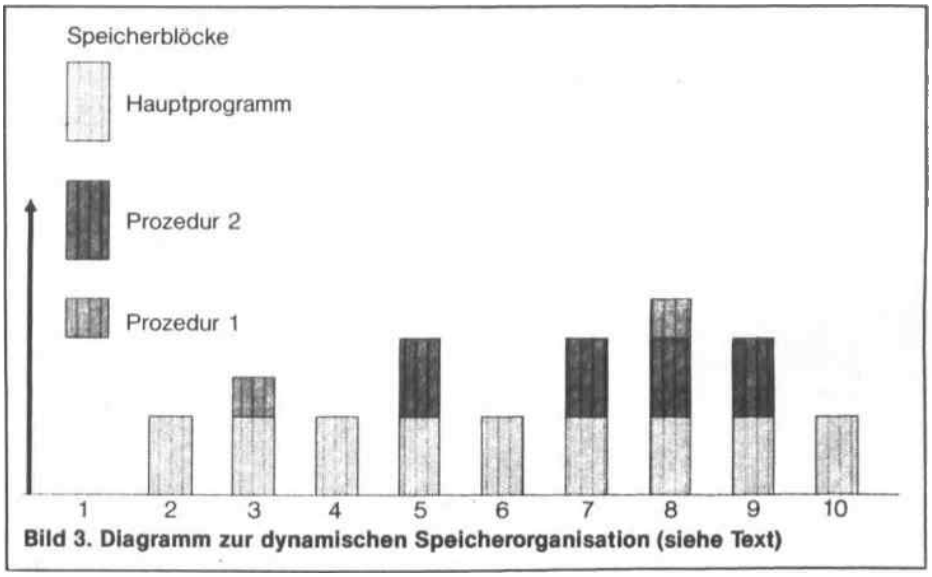


Bild 3. Diagramm zur dynamischen Speicherorganisation (siehe Text)

```
1/3   = 0.13
1/12  = 0.08^3
1/17 = 0.10588235294117647
```

Wie findet man aber den Anfang (und damit auch das Ende) der Periode? Zunächst erinnern wir uns an die Rechnung mit Brüchen aus der Schule:

```
1.000000 / 7 = 0.1142857
1 0
  30
   20
    60
     40
      50
       10
```

Beginnend mit dem REST 1 multipliziert man immer 10 und dividiert das Produkt wieder durch 7. Der ganzzahlige Anteil REST DIV 7 ist die nächste Ziffer des Ergebnisses, während der Divisionsrest REST MOD 7 den REST für den nächsten Divisionsschritt bildet.

Die Periode endet genau dann, wenn der REST bereits früher einmal in der Berechnung (nicht unbedingt als erster, siehe 1/12!) aufgetreten ist. Daher speichert die Prozedur alle Ziffern in einem Array ZIFFERN und für jeden Rest R den Index der zugehörigen Ziffern in dem Array INDEX. Wird dabei festgestellt, daß dieser INDEX ungleich 0 ist, also dieser Rest bereits einmal auftrat, so ist die gesamte Periode bekannt, und das Ergebnis kann anschließend gedruckt werden. Probieren Sie es doch einmal mit 1/29!

Zum Verständnis der Prozedur Listing 11 benötigen Sie noch Kenntnisse über die Stringbehandlung in Pascal. Bisher wurden in allen Beispielprogrammen nur einzelne Zeichen, also Werte vom Typ CHAR, verwendet. Möchte man jedoch Worte, Zeilen oder Sätze, die aus einer Folge von Zeichen bestehen, verarbeiten, so muß man ein Array aus Zeichen definieren.

```
CONST TEXT='Otto Anna';
TYPE  STRING = ARRAY [1..9]
OF CHAR;
VAR   WORT : STRING;
      ZEILE: ARRAY [1.80]
OF CHAR;
```

Text ist eine Konstante mit dem Typ ARRAY [1..9] OF CHAR, da sie aus neun Zeichen besteht. Denselben Typ besitzt die Variable Wort. Die Variable Zeile umfaßt 80 einzelne Zeichen. Wichtig ist jetzt die Tatsache, daß jedem String eine feste Länge zugeordnet ist. Es ist also nicht möglich, in Wort nur vier Zeichen zu speichern, man muß vielmehr den Rest des Strings zum Beispiel mit Leerzeichen füllen:

```
WORT:= 'ANNA_____'; (korrekt)
WORT:= 'ANNA';      (falsch)
WORT:= TEXT;        (korrekt)
WORT:= ZEILE;       (falsch)
Zeile:= WORT;       (ebenfalls
                     falsch!)
```

```
FUNCTION SIGNUM (X:REAL): INTEGER;
(* Vorzeichen: 0, -1 oder +1 *)

BEGIN
  IF X>0 THEN SIGNUM:= 1
  ELSE
    IF X<0 THEN SIGNUM:= -1
    ELSE SIGNUM:= 0
END; (* SIGNUM *)

FUNCTION HOCH (X,Y: REAL):REAL;
(* ERGEBNIS IST X ^ Y *)
BEGIN
  HOCH:= EXP( Y * LOG(X));
END; (* HOCH *)

FUNCTION FAKULTAET (N:INTEGER):REAL;
(* ERGEBNIS IST N ! = 1 * 2 * 3 * 4 *
... * N  *)
            VAR P:  REAL;
                I:  INTEGER;
BEGIN
  P:= 1.0;
  FOR I:= 2 TO N DO P:= P * I
  FAKULTAET:= P
END; (* FAKULTAET *)

GEOMETRISCHES_MITTEL(A,B:REAL): REAL;
BEGIN
  GEOMETRISCHES_MITTEL:= SQRT(A*B)
END;

ARITHMETISCHES_MITTEL(A,B,C:REAL):
REAL; BEGIN
  ARITHMETISCHES_MITTEL:=
  (A + B + C) / 3
  END;
```

Listing 8. Einige Beispiele für Funktionen in Pascal

Zuweisungen sind also nur zwischen Strings gleicher Länge möglich, da weder überflüssige Zeichen abgeschnitten noch zu kurze Strings mit Leerzeichen erweitert werden. Andererseits kann man jedoch auch Strings gleicher Länge miteinander vergleichen. Dabei bestimmt das Ergebnis des Vergleiches den zugrundeliegenden Zeichensatz. Im ASCII-Code gilt beispielsweise

```
'ALPHA' <'BETA '
'ALPHA' <'alpha'
'ALP  ' <'ALPHA'
```

Die hier gemachten Einschränkungen gelten für Standard-Pascal. Für Turbo-Pascal und andere Compiler gelten sie nicht.

Die Funktion POSITION in Listing 11 liefert die Position von Wort in der Tabelle TAB. Diese Tabelle soll aufsteigend mit Strings gefüllt sein. Aus der Funktion resultiert der Wert 0, falls Wort nicht in TAB enthalten ist.

```
CONST N = 100;

TYPE  SORTIERELEMENT = REAL;
      SORTIERFELD    = ARRAY (UG..OG)
      OF SORTIERELEMENT;

PROCEDURE SORT(VAR A: SORTIERFELD);

  VAR I,J,K: INTEGER;
      MAX  : SORTIERELEMENT;

BEGIN
  FOR J:= OG DOWNTO UG+1 DO
    BEGIN
      MAX:= A[UG]; K:= UG;
      FOR I:= UG+1 TO J DO
        IF A[I] > MAX THEN
          BEGIN
            K:= I; MAX:= A[I]
          END;
      A[K]:= A[J]; A[J]:= MAX
    END;
END; (* SORT *)
```

Listing 9. Sortieren durch Auswahl

Falls Sie einmal ein Programm schreiben möchten, das Ihnen alle Namen in einem Pascal-Programm ausdruckt, so müssen Sie zunächst alle Schlüsselworte erkennen. Hierzu könnten Sie die Funktion POSITION verwenden. In TAB werden alle Schlüsselworte in alphabetischer Reihenfolge eingetragen (AND, ARRAY, BEGIN, . . .). Dann ergeben sich folgende Funktionsergebnisse:

```
POSITION('AND       ',TAB) = 1
POSITION('BEGIN     ',TAB) = 3
POSITION('WITH      ',TAB) = 39
POSITION('WIRSING   ',TAB) = 0
```

Obwohl die Funktion etwas komplizierter aussieht, arbeitet sie für große Arrays wesentlich schneller als die einfache Version aus Listing 12, die das Array schrittweise von hinten nach vorne durchsucht. Statt dessen verwendet die erste Funktion zwei Zeiger R und L auf den rechten und linken Rand des Teilarrays, in dem sich der gesuchte Wert befinden muß. In jedem Schritt wird nun der Mittelpunkt M des Intervalls untersucht. Steht an dieser Stelle ein Wort, das größer als das Suchwort ist, so liegt das Suchwort im Intervall L bis M-1, ansonsten muß es im Interval M+1 bis R zu finden sein. Die Suche ist beendet, falls die Zeiger L und R sich überschneiden. Offensichtlich wird in jedem Schritt das Array halbiert, was den Geschwindigkeitsvorteil gegenüber Listing 12 ausmacht.

Noch ein Wort zu der linearen Suche in Listing 12: Warum wurde nicht einfach die Schleife

```
I:= ANZAHL;
WHILE (WORT <>TAB[I] AND
(I <>0) DO I :=I-1;
POSITION:= I
```

verwendet? Ist in diesem Fall Wort nicht in TAB vorhanden, so ist im letzten Schleifendurchlauf I=0, was im ersten Teil der WHILE-Bedingung einen Zugriff auf das (nicht existierende) Element TAB[0] zur Folge hat. Normalerweise erkennt ein Pascal-Laufzeitsystem solche Indizierungsfehler und gibt dann eine Fehlermeldung aus.

Rekursion

Das letzte Programm in Listing 13 ist ein besonders schönes Beispiel für eine sehr nützliche Eigenschaft von Prozeduren und Funktionen in Pascal. Falls Sie sich an die Ausführungen über die Speicherverwaltung erinnern, wissen Sie noch, daß bei jedem Prozeduraufruf neuer Speicherplatz für die lokalen Variablen bereitgestellt wird. Des-

halb kann sich eine Prozedur auch selbst aufrufen, ohne daß dadurch der Inhalt der lokalen Variablen zerstört würde. Den Selbstaufruf einer Prozedur oder Funktion nennt man Rekursion. Das Programm in Listing 13 verwendet nun eine rekursive Prozedur PROBIERE_ZEILE, um das »Problem der acht Damen« zu lösen.

Die Aufgabe besteht darin, acht Königinnen, die nach den Schachregeln alle Figuren in ihrer Zeile und Spalte sowie in beiden Diagonalen bedrohen, so auf ein Schachbrett zu setzen, daß keine Figur eine andere bedroht. Bild 4 zeigt die Felder, die eine Dame angreift, während Bild 5 eine zulässige Lösung zeigt.

Die Lösung stellt ein sogenannter Backtracking Algorithmus. Hierbei probiert man eine Reihe von Schritten aus, bis man feststellt, daß dieser Weg in eine Sackgasse führt. In diesem Moment beginnt man die letzte Entscheidung rückgängig zu machen und einen neuen Weg zur Lösung zu finden. Indem man systematisch alle möglichen Kombinationen durchsucht, wird auf jeden Fall eine Lösung (sofern vorhanden) gefunden.

In diesem konkreten Beispiel positioniert man zunächst eine Dame an eine zulässige Stelle der ersten Zeile. Dabei wird notiert, welche Diagonalen und welche Spalte die Dame bedroht. Anschließend wird in der zweiten Reihe eine ungefährdete Position gesucht, die die zweite Dame besetzt. Diese Prozedur wiederholt sich so lange, bis die achte Zeile belegt und die Lösung gefunden ist. Im Normalfall ist natürlich bereits in der dritten oder vierten Zeile jedes Feld gefährdet, so daß die Suche in eine Sackgasse führt. Dann wird die Dame von ihrem Feld wieder entfernt und in der letzten Reihe die Dame auf das nächste freie Feld gesetzt. Mit dieser Strategie erhält man alle 92 Lösungen auf einem 8 x 8-Brett.

Das größte Problem besteht darin, möglichst schnell zu testen, ob eine Diagonale oder eine Spalte bereits belegt ist. Deshalb werden drei boolesche Arrays geführt, die für jede Diagonale und Spalte den Wert TRUE enthalten, falls diese nicht bedroht ist. Natürlich muß die Information in diesem Array am Programmanfang gelöscht und bei jedem Zug und jeder Zugrücknahme aktualisiert werden. Bild 6 zeigt, wie man aus dem Zeilen- und Spaltenindex I,J die Nummer der jeweiligen Diagonalen durch Addition und Subtraktion erhält.

Bisher wurde nur das Array als zusammengesetzter Typ verwendet. Durch die Flexibilität bei der Wahl des Indextyps und des Typs der Elemente konnten sehr verschiedenartige Probleme behandelt werden. Bevor wir uns weiteren Datentypen in Pascal zuwenden, kommen noch zwei weitere Eigenschaften von Arrays zur Sprache.

1. Man kann auch das Array als Ganzes in einer Zuweisung verwenden, falls auf der rechten und linken Seite des Zuweisungsoperators je ein Array desselben Typs steht:

```
VAR A,B: ARRAY[1..100] OF REAL;
A:=B;
```

Mit dieser Zuweisung werden also 100 reelle Zahlen aus dem Array B in das Array A kopiert. Somit enthält diese allgemeine Regel die oben genannten Bedingungen, unter denen eine Zuweisung zwischen Strings zulässig ist.

```
BEGIN
  IF I>MAXIMAL THEN  WRITELN('DIE ZAHL IST ZU GROSS!')
  ELSE
    BEGIN
      WRITE('1 / ', I, ' = 0.');
      (* NOCH IST KEIN REST AUFGETRETEN, DESHALB INDEX LOESCHEN: *)
      FOR J:= 0 TO I-1 DO
        INDEX[J]:= 0;

      K    := 0;       (* DIE ERSTE ZIFFER WIRD BERECHNET *)
      REST:= 1;        (* WIR BERECHNEN   1 / I           *)
      REPEAT
        K:= K+1; INDEX[REST]:= K;
        REST:= ZEHN * REST;
        ZIFFER[K]:= REST DIV I;
        REST     := REST MOD I;
      UNTIL INDEX[REST]<>0;       (* BIS DIESER REST SCHON EINMAL BERECHNET *)

      (* JETZT NOCH DAS ERGEBNIS DRUCKEN, INDEX[REM] IST DIE POSITION DER  *)
      (* ERSTEN ZIFFER IN DER PERIODE                                      *)
      FOR J:= 1 TO INDEX[REST]-1 DO
         WRITE(CHR(ZIFFER[J]+ORD('0')));
      WRITE('...');
      FOR J:=INDEX[REST] TO K DO
         WRITE(CHR(ZIFFER[J]+ORD('0')));
      WRITELN
    END;
END; (* KEHRWERT *)
PROCEDURE KEHRWERT (I: INTEGER);
(* DRUCKE DEN EXAKTEN KEHRWERT DER ZAHL I MIT ANGABE DER PERIODE        *)

   CONST ZEHN = 10;          (* BASIS DES DEZIMALSYSTEMS                *)
         MAXIMAL = 300;      (* MAXIMALE GROESSE FUER I UND DIVISIONSREST *)

   VAR   J,K      : INTEGER;
         REST     : INTEGER;(* LAUFENDER DIVISIONSREST BEI STELLE K     *)

         ZIFFER   : ARRAY[1..MAXIMAL] OF INTEGER;
         INDEX    : ARRAY[0..MAXIMAL] OF INTEGER;
```

Listing 10. Exakte Berechnung des Kehrwertes

```
CONST ANZAHL = 30;                    (* LAENGE DER SUCH-TABELLE          *)

TYPE  STRING = ARRAY[1..11] OF CHAR; (* LÄNGE EINES WORTES = 11 ZEICHEN *)
      TABELLE= ARRAY[1..ANZAHL] OF STRING;

FUNCTION POSITION (WORT: STRING; VAR TAB: TABELLE);
   TAB MUSS AUFSTEIGEND ALPHABETISCH SORTIERT SEIN. DAS ERGEBNIS DER     *)
   FUNKTION IST DIE POSITION VON WORT IN TAB BZW. 0, FALLS WORT NICHT    *)
(* IN TAB STEHT.                                                         *)

  VAR L,R,M: INTEGER;

BEGIN
  L:=1; R:=ANZAHL;
  REPEAT
    M:= (L+R) DIV 2;                  (* MITTELPUNKT DES INTERVALLS L..R *)
    IF WORT<=TAB[M] THEN R:= M-1;     (* WORT LIEGT LINKS VON DER MITTE  *)
    IF WORT>=TAB[M] THEN L:= M+1;     (* WORT LIEGT RECHTS VON DER MITTE *)
  UNTIL L>R;                          (* BIS ZEIGERKOLLISION             *)
  IF L>R+1 THEN POSITION:=R+1;        (* WORT GEFUNDEN                   *)
           ELSE POSITION:=0           (* WORT NICHT VORHANDEN            *)
END; (* POSITION *)
```

Listing 11. Binäre Suche

```
PROCEDURE POSITION (WORT: STRING; VAR TAB: TABELLE);
(* EINFACHE LINEARE SUCHE NACH WORT IN TAB. TAB MUSS NICHT SORTIERT SEIN *)

  VAR I: INTEGER;

BEGIN
  I:=ANZAHL;
  WHILE (TAB[I]<>WORT) AND (I<>1) DO I:=I-1;
  IF TAB[I]=WORT THEN POSITION:=I
                 ELSE POSITION:=0
END; (* POSITION *)
```

Listing 12. Lineare Suche

2. Arrays können als Elemente nicht nur einfache, sondern auch zusammengesetzte Typen enthalten. Wiederum hatten wir diese Tatsache bereits bei der Besprechung der Strings vorweggenommen, als wir definierten:

```
TYPE STRING = ARRAY[1..11]
OF CHAR;
     TABELLE= ARRAY[1..ANZAHL]
OF STRING;
```

Ist der Elementtyp ein Array, so spricht man oft auch von mehrdimensionalen Arrays. In diesen Fällen kann man auch eine vereinfachte Notation verwenden. Statt

```
VAR A,B: ARRAY [1..N] OF
           ARRAY [1..M] OF
             ARRAY [1..K] OF REAL;
A[1][1][1]:= B[2][3][4]
```

schreibt man dann:

```
VAR A,B: ARRAY[1..N,1..M,1..K]
OF REAL;
A[1,1]:=B[2,3,4]
```

Bei der Arbeit mit solchen mehrdimensionalen Tabellen verwendet man sehr häufig Ausschnittstypen. Betrachten Sie zum Beispiel (Bild 7), das eine zweidimensionale Tabelle mit Umsatzzahlen (Millionen DM ?) für fünf Jahre mit jeweils zwölf Monaten enthält. Das kann man schreiben als:

```
TYPE MONAT = 1..12;
     JAHR  = 1980..1985;
VAR UMSATZTABELLE = ARRAY
[JAHR,MONAT]
                        OF REAL;
    M : MONAT;
    J : JAHR;
```

Die Variablen M und J besitzen die Ausschnittstypen MONAT und JAHR und können daher nur die Werte 1 bis 12 oder 1980 bis 1985 annehmen. Sie eignen sich also hervorragend als Zeiger in die zweidimensionale Tabelle. Jede Zuweisung an eine Variable eines Ausschnittstyps wird zur Laufzeit auf gültige Werte geprüft, so daß bei folgenden Anweisungen das Programm mit einer Fehlermeldung endet:

```
J:= 12; M:= 1985
```

Einmal schützt also der Compiler vor irrtümlichen Zuweisungen. Listing 14 zeigt drei Funktionen, die in solchen Umsatztabellen Zeilen-, und Spalten-, oder Gesamtsummen berechnen. Das Hauptprogramm verwendet diese Funktionen für die Ermittlung von Umsätzen für zwei Produkte (Farben und Lacke).

Allgemein kann man von jedem skalaren Typ Ausschnittstypen bilden:

```
TYPE BUCHSTABEN = 'A'..'Z';
     ZIFFERN    = '0'..'9';
```

Eine Variable eines Ausschnittstyps kann überall dort verwendet werden, wo auch eine Variable des zugehörigen Grundtyps (INTEGER oder CHAR) stehen kann. Unter Umständen kann die Verwendung von Ausschnittstypen (zum Beispiel als Elementtyp in großen Arrays) bei einigen Compilern auch Speicherplatz einsparen. Mit der Kenntnis, daß die Variable JAHR nur die Werte 1980 bis 1985 annnimmt, genügt 1 Byte zur Speicherung des Wertes (statt 2 oder 4 Byte für die normalen INTEGER-Zahlen).

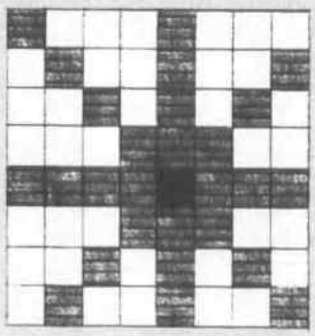


Bild 4. Beispiel für die von einer Dame bedrohten Felder

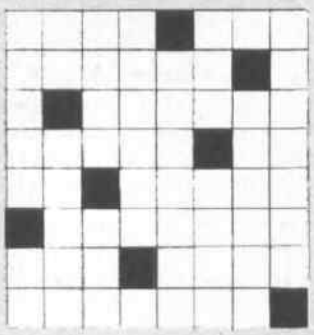


Bild 5. Eine von mehreren Lösungen des Acht-Damen-Problems

	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	8	9	10
3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12
5	6	7	8	9	10	11	12	13
6	7	8	9	10	11	12	13	14
7	8	9	10	11	12	13	14	15
8	9	10	11	12	13	14	15	16

	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7
2	-1	0	1	2	3	4	5	6
3	-2	-1	0	1	2	3	4	5
4	-3	-2	-1	0	1	2	3	4
5	-4	-3	-2	-1	0	1	2	3
6	-5	-4	-3	-2	-1	0	1	2
7	-6	-5	-4	-3	-2	-1	0	1
8	-7	-6	-5	-4	-3	-2	-1	0

Bild 6. Berechnung der Diagonalen im Programm ACHT DAMEN (links: Addition der Zeilen- und Spaltennummern, rechts entsprechend Subtraktion).

Eigene Datentypen definieren

Eine andere Methode, neue Typen zu definieren, besteht darin, daß man alle Werte aufzählt, die dieser Typ annimmt:

```
TYPE AMPEL = (ROT, GELB, GRUEN);
     FAMILIENSTAND = (LEDIG,
                      VERHEIRATET,
                      GETRENNT,
                      GESCHIEDEN,
                      VERWITWET);
     FIGUR         = (BAUER,
                      LAEUFER,
                      SPRINGER,
                      TURM,
                      DAME,
                      KOENIG);
     FARBE         = (SCHWARZ,
                      WEISS);
```

Diese Typen nennt man Aufzählungstypen. Formal gesehen sind zum Beispiel ROT und KOENIG Konstanten des jeweiligen Typs. Durch die Reihenfolge bei der Typendeklaration wird eine Ordnung auf den Konstanten definiert:

```
ORD(ROT) = 0
ORD(GELB) = 1
ORD(GRUEN) = 2
```

Somit sind auch Vergleiche zwischen Variablen eines Aufzählungstyps sinnvoll:

```
VAR HAUPTAMPEL : AMPEL;
IF HAUPTAMPEL <= GELB THEN ...
```

Um den Nachfolger und Vorgänger im Wertebereich zu erhalten, gibt es die Funktionen PRED (predecessor) und SUCC (successor):

```
SUCC(ROT) = GELB
SUCC(LAEUFER) = SPRINGER
PRED(VERHEIRATET) = LEDIG
PRED(WEISS) = SCHWARZ
```

ORD, SUCC und PRED sind übrigens für jeden skalaren Typ zulässig, da alle skalaren Typen geordnet sind:

```
ORD(FALSE)=0
SUCC(FALSE)=TRUE
SUCC('A') = 'B'
PRED(0) = -1
```

Viele hoffnungsvolle Programme von Anfängern enthalten direkte Ein- und Ausgabeanweisungen für Werte von Aufzählungstypen:

```
WRITE (HAUPTAMPEL);
READ (HAUPTAMPEL);
```

Diese Anweisungen sind falsch! Nur während der Übersetzung sind dem Compiler die Namen der Werte des Typs bekannt. Zum Zeitpunkt der Ausführung sind alle Aufzählungstypen durch Zahlen ohne Verweise auf irgendwelche Namen codiert. Man muß also die Umwandlung zwischen dem Aufzählungstyp und dem auszugebenden Namen selbst vornehmen:

```
PROCEDURE DRUCKE_AMPEL(A:AMPEL);
BEGIN
   WRITE('Die Ampel zeigt);
   CASE A OF
    ROT  :  WRITELN('rot');
    GELB :  WRITELN('gelb');
    GRUEN:  WRITELN('grün');
  END;
END; (* DRUCKE_AMPEL *)
```

Natürlich können auch Auschnittstypen von Aufzählungstypen gebildet werden:

```
TYPE ENDSPIELFIGUR =
LAEUFER..KOENIG;
```

Mit den Aufzählungstypen lernen Sie den letzten skalaren Typen in Pascal kennen. Alle weiteren Typen, die den Rest des Artikels einnehmen, sind zusammengesetzte Typen wie das Array. Mengen von Zahlen sind Ihnen sicher bekannt. Um einen Mengentyp zu deklarieren, verwendet man die Wortsymbole SET und OF, denen ein skalarer Typ folgen muß:

```
TYPE FIGUREN = SET OF FIGUR;
VAR MEINE, DEINE: FIGUREN;
```

Mit den Mengen-Variablen MEINE und DEINE kann man beispielsweise über die geschlagenen Figuren führen. Am Spielbeginn gilt:

```
MEINE:= [BAUER...KOENIG];
DEINE:= MEINE;
IF [TURM,DAME,KOENIG]<
=MEINE THEN...
```

Mit dieser Abfrage wird also getestet, ob ein Spieler noch Turm, Dame und König besitzt. Da eine Menge jedes Element nur einmal oder keinmal enthält, werden Mengen überall dort verwendet, wo das Vorhandensein einer gewissen Eigenschaft signalisiert werden soll.

Arrays und Mengen fassen mehrere Werte eines Typs zu einem zusammengesetzen Typ zusammen. Besonders in großen Programmen ist es jedoch auch sinnvoll, Werte verschiedener Typen zu einer Einheit zu verbinden. Zur Konstruktion solcher Typen verwendet man in Pascal Recordtypen (Verbundtypen):

```
TYPE POSITION = RECORD
                  X,Y: REAL;
                END;
ADRESSE  = RECORD
     NAME, VORNAME: STRING;
     ORT, STRASSE : STRING;
     HAUSNUMMER   : INTEGER;
     PLZ          : INTEGER;
  END;
```

```
PROGRAM ACHT_DAMEN (INPUT,OUTPUT);

CONST N       = 8;              (* ACHT DAMEN AUF EINEM 8 * 8 SCHACHBRETT *)
      N2      = 16;             (* = 2 * N *)

VAR   DAME        : ARRAY [1..N]  OF INTEGER;
      SPALTE_FREI : ARRAY [1..N]  OF BOOLEAN;
      DIAG1_FREI  : ARRAY [2..N2] OF BOOLEAN;
      DIAG2_FREI  : ARRAY [-N..N] OF BOOLEAN;
      LOESUNG     : INTEGER;
      I           : INTEGER;

  PROCEDURE DRUCKELOESUNG;
  (* ZEIGE DIE POSITION DER DAMEN AN, ERHÖHE DEN ZÄHLER FÜR DIE LÖSUNGEN *)

    VAR ZEILE, SPALTE: INTEGER;

  BEGIN
    LOESUNG:= LOESUNG + 1;
    WRITELN; WRITELN; WRITELN('LÖSUNG', LOESUNG:2, ':');
    FOR ZEILE:=1 TO N DO
      BEGIN
        FOR SPALTE:=1 TO N DO
          IF SPALTE = DAME[ZEILE] THEN WRITE('*')
                                  ELSE WRITE('+');
        WRITELN;
      END;
  END; (* DRUCKELOESUNG *)

  PROCEDURE SETZE_ZEILE(I: INTEGER);
  (* SUCHE FREIE POSITION FÜR DAME IN ZEILE I. FALLS EIN PLATZ GEFUNDEN *)
  (* WURDE, WIRD EINE DAME IN ZEILE I+1 GESETZT ODER DIE VOLLSTAENDIGE  *)
  (* STELLUNG ANGEZEIGT. SONST ERFOLGT EIN RUECKSPRUNG.                 *)

    VAR J: INTEGER;

  BEGIN
    FOR J:= 1 TO N DO
      IF SPALTE_FREI[J] AND DIAG1_FREI[I+J] AND DIAG2_FREI[J-I] THEN
        BEGIN (* HIER KANN JETZT DIE DAME GESETZT WERDEN: *)
          DAME[I]:= J;
          SPALTE_FREI [J] := FALSE;
          DIAG1_FREI [I+J]:= FALSE;
          DIAG2_FREI [J-I]:= FALSE;

          IF I=N THEN (* ALLE DAMEN GESETZT, LÖSUNG DRUCKEN! *)
            DRUCKELOESUNG
          ELSE        (* IN DER NÄCHSTEN ZEILE WEITERSUCHEN *)
            SETZE_ZEILE(I+1);

          (* JETZT DEN LETZEN ZUG RÜCKGÄNGIG MACHEN:         *)
          SPALTE_FREI [J] := TRUE;
          DIAG1_FREI [I+J]:= TRUE;
          DIAG2_FREI [J-I]:= TRUE;
        END;
  END; (* SETZE_ZEILE *)

BEGIN
  (* SPIELBRETT IST NOCH LEER: *)
  FOR I:= 1 TO 8  DO SPALTE_FREI [I]:= TRUE;
  FOR I:= 2 TO N2 DO DIAG1_FREI  [I]:= TRUE;
  FOR I:=-N TO N  DO DIAG2_FREI  [I]:= TRUE;
  LOESUNG:= 0;
  SETZE_ZEILE(1);
END.
```

Listing 13. Das Acht-Damen-Problem, ein Beispiel für einen rekursiven Algorithmus.

	Monat											
	1	2	3	4	5	6	7	8	9	10	11	12
1980	1.1	2.1	3.4	5.5	6.6	0.4	3.1	0.0	8.2	9.9	0.0	3.2
1981	1.2	2.3	3.4	4.9	6.2	1.4	3.2	0.1	7.9	9.5	0.2	3.3
1982	1.1	1.9	3.2	4.5	6.3	1.7	3.4	0.2	8.0	9.5	0.0	3.1
1983	1.4	2.2	3.3	4.3	6.6	2.3	3.5	0.3	8.1	9.5	0.2	3.0
1985	1.8	2.3	3.4	4.1	6.7	2.4	3.6	0.4	8.3	9.3	0.0	3.5

Bild 7. Darstellung von Jahresumsätzen als Beispiel

```
VAR P1,P2 : POSITION;
    ADR   : ADRESSE;
```

Nach einer solchen Deklaration bezeichnet man P1, P2 und ADR als Records. Ein Record besitzt einzelne Felder, auf die man durch Angabe ihres Namens nach einem Punkt zugreift:

```
P1.X:= 30;
P1.Y:= 40;
ADR.ORT     := 'MONOPOLY';
ADR.STRASSE:= 'Schloßallee';
ADR.HAUSNR := 4;
```

Andererseits kann man aber auch den Record als Ganzes ansprechen:

```
P1:= P2
```

Durch diese Zuweisung werden alle

```
PROGRAM UMSAETZE (INPUT, OUTPUT);

  TYPE MONAT = 1..12;
       JAHR  = 1980..1985;
       UMSATZTABELLE: ARRAY [JAHR,MONAT] OF REAL;

  VAR  FARBEN, LACKE: UMSATZTABELLE;

  FUNCTION JAHRESSUMME( J: JAHR; VAR T: UMSATZTABELLE): REAL;
  (* BERECHNE DIE SUMME DER MONATSUMSÄTZE IM JAHR J IN DER
  TABELLE T: *)

   VAR M: MONAT;
       S: REAL;

BEGIN
  S:= 0;
  FOR M:= 1 TO 12 DO S:= S + T[J,M];
  JAHRESSUMME:= S
END; (* JAHRESSUMME *)

FUNCTION MONATSSUMME( M: MONAT; VAR T: UMSATZTABELLE): REAL;
(* BERECHNE DIE SUMME ALLER UMSÄTZE IM MONAT M FÜR 1980
BIS 1985 *)
   VAR J: JAHR;
       S: REAL;

  BEGIN
    S:= 0;
    FOR J:= 1980 TO 1985 DO S:= S + T[J,M];
    MONATSSUMME:= S
  END; (* MONATSSUMME *)

  PROCEDURE GESAMTSUMME (VAR T: UMSATZTABELLE): REAL;
   (* BERECHNE DIE SUMME ALLER UMSÄTZE *)

    VAR J: JAHR;
        S: REAL;

  BEGIN
    S:= 0;
    FOR J:= 1980 TO 1985 DO S:= S + JAHRESSUMME(J,T);
    GESAMTSUMME:= S
  END; (* GESAMTSUMME *)

BEGIN

  (* HIER WERDEN DIE UMSÄTZE FÜR FARBEN UND LACKE VORBELEGT *)

  WRITELN('UMSATZ FARBEN 1980:', JAHRESSUMME(1980, FARBEN));
  WRITELN('UMSÄTZE LACKE JANUAR 1980-1985:'); MONATSSUME(1,
  LACKE); WRITELN('UMSÄTZE LACKE UND FARBEN INSGESAMT:',
          GESAMTSUMME(LACKE) + GESAMTSUMME (FARBEN));
END.
```

Listing 14. Bearbeitung zweidimensionaler Tabellen

Felder im Record übertragen. Wichtig und charakteristisch für das Typkonzept in Pascal ist die Tatsache, daß zusammengesetzte Typen auch mehrstufig aufzubauen sind:

```
VAR POSITIONEN: ARRAY [1...99] OF
                          POSITION;
```

Den umgekehrten Fall, daß ein Record aus Arrays besteht, finden Sie bereits im Beispiel des Typs ADRESSE. Dort sind die Felder NAME und VORNAME vom Typ STRING, den wir ja als

```
TYPE STRING = ARRAY [1..11]
OF CHAR;
```

vereinbart hatten. Sollen viele Operationen mit den Feldern eines Records durchgeführt werden, so ist die ständige Wiederholung des Variablennamens lästig. Für diesen Fall existiert die WITH-Anweisung:

```
WITH Recordvariable DO Anweisung
```

So kann man folgende Zuweisungen

```
ADR.NAME    :='Hugendubel';
ADR.VORNAME:='Kunigunde';
ADR.ORT     :='7 Berge';
ADR.STRASSE:='7 Zwerge';
```

schreiben als:

```
WITH ADR DO
BEGIN
   NAME    := 'Hugendubel';
   VORNAME:= 'Kunigunde';
   ORT     := '7 Berge';
   STRASSE:= '7 Zwerge';
END;
```

Wie alles in Pascal können auch WITH-Anweisungen geschachtelt werden. Nach der geschachtelten Record-Deklaration

```
TYPE DATUM = RECORD
               TAG    :  1..31;
               MONAT  :  1..12;
               JAHR   :  INTEGER;
             END;
BANKAUSZUG = RECORD
         DATUM: DATUM;
         KONTOSTAND: REAL
         END;
VAR MEINBANKAUSZUG: BANKAUSZUG;
```

prüft man folgendermaßen das Datum auf dem Bankauszug:

```
WITH MEINBANKAUSZUG DO
BEGIN
 IF KONTOSTAND >0 THEN
  WITH DATUM DO
   WRITELN(TAG,'.',MONAT,'.',
JAHR)
END;
```

Dabei muß sich die innere WITH-Anweisung auf ein Feld in dem Record der äußeren WITH-Anweisung beziehen. Hier ist dies also das Feld MEIN BANKAUSZUG.DATUM. Bemerkenswert sind noch die Sichtbarkeitsregeln für Feldnamen. Man kann ohne Namenskonflikte Variablen mit dem Feldnamen aus der Record-Deklaration definieren. Der Feldname ist durch einen Punkt oder die WITH-Anweisung an den Variablennamen des Records gebunden.

Am besten verdeutlicht wiederum ein Beispiel die Arbeit mit Records. In Listing 15 wird der Typ

```
TYPE BRUCH = RECORD
       ZAEHLER, NENNER: INTEGER;
     END;
```

definiert. Man möchte also auch gebrochene Zahlen »exakt« darstellen. Nun lernt man aber in der Schule, daß

$\frac{1}{2}$ und $\frac{2}{4}$ und $\frac{123456}{245912}$

dieselbe Zahl darstellen. Deshalb werden bei allen Operationen Zähler und Nenner gekürzt dargestellt. Zum Kürzen muß man den größten gemeinsamen Teiler (ggT) von Zähler und Nenner kennen. Beim Addieren werden die beiden Brüche auf einen Hauptnenner gebracht, der natürlich möglichst klein bleiben soll. Daher ist im Programm auch eine Funktion zur Berechnung des kleinsten gemeinsamen Vielfachen (kgV) enthalten. Nach der Erweiterung der beiden Zähler ergibt sich durch Addieren der Zähler des Summenbruches. Wegen der teilerfremden Ausgangsbrüche und der Wahl des Hauptnenners spart man es sich, den Zähler gegen den Nenner zu kürzen.

Die Multiplikation verläuft wie in der Schule nach der Regel Zähler mal Zähler und Nenner mal Nenner. Jedoch werden zuvor die Brüche kreuzweise gekürzt, um die Produkte möglichst klein zu halten. Vielleicht ist es eine gute Denksportaufgabe, zu überlegen, warum nach der Multiplikation Zähler und Nenner teilerfremd sind.

Die Subtraktion ist einfach auf die Addition zurückzuführen, ebenso wie die Division durch Kehrwertbildung auf die Multiplikation zurückführt. Bei der Berechnung des Kehrwertes ist jedoch auf eine korrekte Behandlung des Vorzeichens zu achten.

Normalerweise eignen sich zur Erläuterung von Records Beispiele aus der kaufmännischen Datenverarbeitung, da man einen Record am einfachsten als ein Formblatt beschreiben kann, das verschiedene Felder enthält, die nur Werte gewisser Typen beinhalten dürfen.

```
TYPE KRAFTFAHRZEUGSCHEIN =
  RECORD
    WAGEN: KENNZEICHEN;
    WOHNORT, STANDORT: ADRESSE;
    LEISTUNG: INTEGER;
  END;
```

Diese Deklaration setzt natürlich voraus, daß zuvor die Typen KENNZEICHEN und ADRESSE (zum Beispiel selbst als Records) definiert wurden.

Es gibt jedoch auch Felder, die nur dann gültig sind, falls in einem anderen Feld ein bestimmter Wert steht.

```
TYPE PERSONALAKTE =
   RECORD
     NAME    :  STRING;
     VORNAME:  STRING;
     CASE STAND: FAMILIENSTAND OF
       LEDIG:  ();
       VERHEIRATET,GETRENNT:
         (HEIRAT: DATUM);
       GESCHIEDEN:
         (SCHEIDUNG: DATUM;
          ALIMENTE : BOOLEAN);
   END;
VAR AKTE1: PERSONALAKTE;
```

Dieses Beispiel zeigt einen varianten (veränderlichen) Record. Er besteht aus einem festen Teil (den Feldern NAME und VORNAME), dem ein veränderbarer Teil folgt. Dieser Teil wird durch das Schlüsselwort CASE eingeleitet. Ihm folgt ein Feldname mit Typangabe. In Abhängigkeit der Werte dieses skalaren Typs besitzen die nachfolgenden Feldlisten in runden Klammern Gültigkeit.

Nach der Zuweisung »AKTE1.STAND: = LEDIG« sind nur die Felder NAME und VORNAME gültig. Ist jedoch »AKTE1.STAND = VERHEIRATET«, so wird zusätzlich das Feld HEIRAT vom Typ DATUM relevant. Ihm darf man jetzt Werte des korrekten Typs zuweisen:

```
AKTE1. HEIRAT.TAG  :=7;
AKTE1. HEIRAT.MONAT:=6;
AKTE1. HEIRAT.JAHR :=1973;
```

Dieselben Felder gelten auch für »STAND = GETRENNT«. Sollte im Laufe der Ehegeschichte eine Scheidung eintreten, so werden sich auch in der Personalakte gravierende Änderungen

```
PROGRAM BRUECHE (INPUT, OUTPUT);
(* RECHNUNG MIT BRUECHEN IN DER DARSTELLUNG ZAEHLER, NENNER *)
TYPE BRUCH = RECORD
               ZAEHLER: INTEGER;
               NENNER : INTEGER;
               (* ZAEHLER UND NENNER IMMER     *)
               (* TEILERFREMD, NENNER POSITIV *)
             END;
VAR A, B, SUMME, DIFFERENZ, PRODUKT, QUOTIENT: BRUCH;

  PROCEDURE KUERZE(VAR A,B: INTEGER);
  (* KUERZE A UND B DURCH IHREN GROESSTEN GEMEINSAMEN TEILER *)
    VAR X: INTEGER;

    FUNCTION GGT (X,Y: INTEGER): INTEGER;
    (* BERECHNE DEN GRÖSSTEN GEMEINSAMEN TEILER VON X UND Y *)
      VAR H: INTEGER;
    BEGIN
      IF Y>X THEN
        BEGIN H:=X; X:=Y; Y:=H END;
      REPEAT
        H:= X MOD Y;
        X:= Y; Y:= H
      UNTIL H = 0;
      GGT:= X;
    END; (* GGT *)

  BEGIN (* KÜRZE *)
    X:= GGT(ABS(A),B);
    IF X<>1 THEN
      BEGIN
        A:= A DIV X;
        B:= B DIV X;
      END;
  END; (* KÜRZE *)

  FUNCTION KGV(X,Y: INTEGER): INTEGER;
  (* BERECHNE DAS KLEINSTE GEMEINSAME VIELFACHE VON X UND Y *)

    VAR U,V: INTEGER;

  BEGIN
    U:= X; V:= Y;
    WHILE X<>Y DO
      IF X>Y THEN
        BEGIN X:= X-Y; U:= U+V END
      ELSE
        BEGIN Y:= Y-X; V:= V+U END;

    KGV:= (U+V) DIV 2
    (* ÜBRIGENS IST GGT(X,Y) = X *)
  END; (* KGV *)

PROCEDURE ADDIERE (A,B: BRUCH; VAR C: BRUCH);
(* ADDIERE DIE BRÜCHE A UND B ZUM BRUCH C *)

   VAR HN: INTEGER; (* HAUPTNENNER *)

BEGIN
  WITH C DO
    BEGIN
      (* HAUPTNENNER BERECHNEN *)
      HN := KGV(A.NENNER, B.NENNER);
      NENNER := HN;
      (* AUF HAUPTNENNER BRINGEN *)
      ZAEHLER := A.ZAEHLER * (HN DIV A.NENNER) + B.ZAEHLER *
                 (HN DIV B.NENNER);
    END; (* WITH *)
END; (* ADDIERE *)

PROCEDURE SUBTRAHIERE(A,B: BRUCH; VAR C: BRUCH);
(* SUBTRAHIERE B VON A. ERGEBNIS IN C *)

BEGIN
  B.ZAEHLER:= - B.ZAEHLER;
  ADDIERE(A,B,C)
END; (* SUBTRAHIERE *)

PROCEDURE MULTIPLIZIERE(A,B: BRUCH; VAR C: BRUCH);
(* ZAEHLER MAL ZAEHLER UND NENNER MAL NENNER. VOR DER *)
(* MULTIPLIKATION WERDEN ZAEHLER UND NENNER GEKUERZT. *)

BEGIN
  (* KUERZE ZUNAECHST KREUZWEISE, DAMIT PRODUKT NICHT *)
  (* ZU GROSS WIRD                                    *)
  KUERZE (A.ZAEHLER, B.NENNER);
  KUERZE (B.ZAEHLER, A.NENNER);

  C.ZAEHLER:= A.ZAEHLER * B.ZAEHLER;
  C.NENNER := A.NENNER  * B.NENNER;

  (* ZÄHLER UND NENNER VON C SIND HIER NOCH TEILERFREMD *)
END; (* MULTIPLIZIERE *)

PROCEDURE KEHRWERT(VAR A: BRUCH);
(* BERECHNE DEN KEHRWERT VON A. BEACHTE DIVISION DURCH 0
UND VORZEICHEN *)

  VAR T: INTEGER;

BEGIN
  T:= A.ZAEHLER;
  IF T=0 THEN WRITELN(' DIVISION DURCH NULL!')
    ELSE
      IF T>0 THEN
        BEGIN
          A.ZAEHLER:= A.NENNER;
          A.NENNER := T
        END
      ELSE
        BEGIN
          A.ZAEHLER:= -A.NENNER;
          A.NENNER := -T;
        END;
END; (* KEHRWERT *)

PROCEDURE LESEN (VAR A: BRUCH);
(* LIEST EINEN BRUCH VON DER TASTATUR. WIRD NACH *)
(* DEM ZAEHLER DAS ZEILENENDE ERREICHT, SO WIRD  *)
(* DER NENNER GLEICH EINS GESETZT                *)

 BEGIN
   WITH A DO
     BEGIN
       READ(A.ZAEHLER);
       IF EOLN THEN A.NENNER:= 1
               ELSE READLN(A.NENNER);
     END;
   KUERZE(A.ZAEHLER,A.NENNER);
 END; (* LESEN *)

 PROCEDURE DRUCKEN (A:BRUCH);

 BEGIN
   WITH A DO
     IF NENNER = 1 THEN WRITELN(ZAEHLER)
     ELSE
       WRITELN(ZAEHLER, '/', NENNER)
 END; (* DRUCKEN *)

BEGIN
 WRITELN('RECHNEN MIT BRUECHEN: (ENDE MIT A = 0)');

 REPEAT
   WRITE('A = '); LESEN(A);
   WRITE('B = '); LESEN(B);

   ADDIERE(A,B,SUMME);
   WRITE('A + B = '); DRUCKEN(SUMME);
   SUBTRAHIERE(A,B,DIFFERENZ);
   WRITE('A - B = '); DRUCKEN(DIFFERENZ);
   MULTIPLIZIERE(A,B,PRODUKT);
   WRITE('A * B = '); DRUCKEN(PRODUKT);
   KEHRWERT(B); MULTIPLIZIERE(A,B,QUOTIENT);
   WRITE('A / B = '); DRUCKEN(QUOTIENT);
 UNTIL A.ZAEHLER = 0;
END.
```

Listing 15. Bruchrechnung mit Records in Pascal

vollziehen. Mit »AKTE1.STAND := GESCHIEDEN« wird das Feld HEIRAT ungültig und statt dessen das Feld SCHEIDUNG (ebenfalls ein Datum) mit dem booleschen Feld ALIMENTE wirksam.

Offensichtlich kann man mit varianten Records sehr gut die Tatsache wiedergeben, daß gewisse Attribute eines Objektes nur unter gewissen Randbedingungen relevant sind. Dies wird dadurch erreicht, daß zu jedem Zeitpunkt immer nur eine der Feldlisten in Klammern hinter den Konstanten gültig ist. Diese zusätzlichen Informationen nutzt der Pascal-Compiler, um wiederum Speicherplatz zu sparen. Er legt alle Varianten (also alle Feldlisten, die zu verschiedenen Werten des Auswahlfeldes gehören) an dieselbe Speicherposition. Zur Laufzeit bestimmt dann das Auswahlfeld nach CASE (engl. tagfield), wie der Inhalt des Speichers zu interpretieren ist. Alle gemeinsamen Felder aus dem festen Teil sind eindeutig, während die Felder im varianten Teil sich gegenseitig überlappen.

Diese Überschneidung muß beachtet werden, wenn ein Wechsel des Auswahlfeldes anfällt. Eine Anwendung varianter Records zeigt Listing 16. Bekanntermaßen kann man einen Punkt in der Ebene auf zwei verschiedene Methoden darstellen:

In rechtwinkligen X- und Y-Koordinaten oder mit Polarkoordinaten, durch Angabe der Entfernung des Punktes zum Koordinatensprung (L) und des Winkels zur X-Achse (PHI).

Da sich eine Darstellungsform für gewisse Anwendungen jeweils besser eignet, wird in diesem Programm mit varianten Records ein Wechsel zwischen beiden Darstellungen zugelassen:

```
TYPE KOORDINATE =
   RECORD
    CASE TYP:KOORDINATENTYP OF
       RECHTWINKLIG:
           (X,Y: REAL);
       POLAR:
           (L,PHI: REAL);
   END;
```

Es ist zu beachten, daß die Angabe des Typs des Auswahlfeldes (hier KOORDINATENTYP) durch einen Typnamen erfolgen muß. Je nach dem Wert von Typ ist also die Koordinate durch X und Y oder L und PHI bestimmt. Im übrigen Programm können Sie neben dem Nutzen der WITH-Anweisung bei der Arbeit mit Records auch den Vorteil der Case-Anweisung bei varianten Records erkennen. Indem man vor dem Zugriff auf eine Variante eine CASE-Anweisung setzt, die die Konstanten aus der Record-Deklaration wiederholt, ist man vor unerlaubten Zuweisungen wie

```
P.TYP:= POLAR;
P.X  := 39.4;
```

geschützt. Erwähnenswert ist noch die Ende-Bedingung der Repeat-Schleife

```
UNTIL BETRAG(P) <= EPSILON;
```

Das Programm soll beendet werden, wenn die Entfernung des Punktes vom Ursprung Null ist. Da jedoch alle reellen Zahlen nur mit Rundungsfehlern berechnet werden können, verwendet man zwischen reellen Zahlen nie den Test auf Gleichheit (A=B), sondern nur eine Prüfung der Form ABS(A-B) <= EPSILON mit einer Konstanten EPSILON, die eine Schranke für die Rundungsfehler auf dem jeweiligen Rechner angibt. Die Berücksichtigung dieser Regel hilft eine Reihe häufiger Fehler zu vermeiden.

Nun sind wir am Ende unserer Einführung in das Programmieren mit Funktionen, Prozeduren und Datentypen angelangt. Zur Vertiefung der gewonnenen Erkenntnisse sei es dringend angeraten, ein wenig mit den abgedruckten Beispielen herumzuexperimentieren, sie zu erweitern und abzuändern. Denn auch in Pascal gilt der altbekannte Spruch vom Meister, den man nur durch Übung . . .

(Florian Matthes/er)

```
PROGRAM KOORDINATEN (INPUT, OUTPUT);
(* RECHNUNG MIT POLAR- UND RECHTWINKLIGEN KOORDINATEN,        *)
(* DARGESTELLT DURCH VARIANTE RECORDS IN PASCAL.              *)
CONST EPSILON = 1.0E-7;                (* RUNDUNGSFEHLERGRENZE *)

TYPE  KOORDINATENTYP = (RECHTWINKLIG, POLAR);
      KOORDINATE = RECORD
                     CASE TYP: KOORDINATENTYP OF
                        RECHTWINKLIG: (X,Y  : REAL);
                        POLAR       : (L,PHI: REAL);
                     END;

VAR  P: KOORDINATE;

PROCEDURE HOLEPUNKT (VAR P: KOORDINATE);

BEGIN
  REPEAT
    WRITE('C-ARTESISCH ODER P-OLAR ?');
    READLN(CH)
  UNTIL CH IN ['C','P'];

  WITH P DO
    IF CH = 'C' THEN
      BEGIN
        TYP:= RECHTWINKLIG;
        WRITE('X = '); READLN(X);
        WRITE('Y = '); READLN(Y)
      END
    ELSE
      BEGIN
        TYP:= CARTESISCH;
        WRITE('L   = '); READLN(L);   (* ABSTAND VOM KOORDINATEN
        *) URSPRUNG
       WRITE('PHI = '); READLN(PHI); (* WINKEL ZUR X-ACHSE *)
     END
END; (* HOLEPUNKT *)

PROCEDURE UMRECHNUNG (VAR P: KOORDINATE);
(* UMRECHNUNG DER BEIDEN KOORDINATENTYPEN *)
  VAR R: REAL;

BEGIN
  WITH P DO
    CASE TYP OF
      RECHTWINKLIG:
        BEGIN
          R  := SQRT(X*X + Y*Y);
          PHI:= ARCTAN(Y/X);
          L  := R;
          TYP:= POLAR;
        END;
      POLAR;
        BEGIN
          R  := SIN(PHI)*L;
          Y  := COS(PHI)*L;
          X  := R;
          TYP:= RECHTWINKLIG
        END;
    END; (* CASE & WITH *)
END; (* UMRECHNUNG *)

FUNCTION BETRAG( P: KOORDINATE): REAL;
BEGIN
  IF P.TYP = RECHTWINKLIG THEN
    BETRAG:= SQRT( SQR(P.X) + SQR(P.Y) )
  ELSE
    BETRAG:= P.L
END; (* BETRAG *)

PROCEDURE DRUCKEPUNKT (P: KOORDINATE);
BEGIN
  WITH P DO
    CASE TYP OF
       RECHTWINKLIG: WRITELN('X = ', X, ', Y = ', Y);
       POLAR       : WRITELN('L = ', L, ', PHI = ',PHI);
    END; (* CASE *)
END; (* DRUCKEPUNKT *)
BEGIN
  REPEAT
    HOLEPUNKT(P);
    DRUCKEPUNKT(P);
    UMRECHNUNG(P);
    DRUCKEPUNKT(P);
  UNTIL BETRAG(P)<= EPSILON;
END.
```

Listing 16. Koordinatenumrechnung mit varianten Records

Dateiverwaltung mit Pascal

Dieser Beitrag beschäftigt sich mit dem Datei-Begriff von Pascal und gibt Hinweise und Beispiele für die effiziente Verwaltung großer Datenmengen.

Hat man große Datenmengen zu verarbeiten, so reicht der Arbeitsspeicher des Computers im allgemeinen nicht aus. Außerdem gehen alle Daten im Hauptspeicher beim Ausschalten des Computers verloren. Um langfristig große Datenbestände zu konservieren, benötigt man externe Speichermedien (Floppy-Disk, Festplatten, aber auch Band- und Kassettenlaufwerke).

Ein großes Problem ist nun die Tatsache, daß es fast so viele Betriebssysteme wie Computertypen gibt. Da jedoch Pascal sowohl auf Mikrocomputern als auch Großrechenanlagen implementiert ist, muß ein möglichst systemunabhängiges Konzept zur Darstellung externer Speicher in der Sprache gefunden werden. Diese Lösung besteht in der Formalisierung des Prinzips der sequentiellen Dateien, die man als den kleinsten gemeinsamen Nenner aller Dateiverwaltungssysteme bezeichnen kann.

Mit der Deklaration

```
TYPE ZAHLENDATEI = FILE OF
INTEGER;
VAR D: ZAHLENDATEI;
```

wird ein File D vereinbart, dessen Komponenten aus ganzen Zahlen bestehen. Allgemein kann nach den Schlüsselworten FILE und OF jeder beliebige (auch zusammengesetzte) Typ folgen. Wie in einem Array besitzen alle Komponenten denselben Typ. Jedoch kann ein File (praktisch) beliebig viele Komponenten umfassen, die nur in fester Reihenfolge adressiert werden können. Zu jedem Zeitpunkt ist immer nur eine Komponente erreichbar. Diese aktuelle Komponente wird im obigen Beispiel mit D↑ bezeichnet und ist vom Typ INTEGER.

Um nun weitere Komponenten zu erreichen, kann man die aktuelle Komponente (Puffervariable) D↑ wie ein Fenster über das gesamte File verschieben (siehe Bild 1). Ein konkretes Beispiel für das Schreiben und Lesen eines Files zeigt Listing 1. Es demonstriert die beiden Methoden des Zugriffs auf Files in Pascal.

1. Sequentielles Schreiben

Um ein File zu erzeugen, muß man die Standardprozedur REWRITE mit der Filevariablen (im Beispiel also D↑) aufrufen. Dadurch werden alle Komponenten gelöscht, die eventuell zuvor im File existierten. Das Schreibfenster (D↑ im Beispiel) steht über der ersten Komponente und besitzt einen undefinierten Wert (siehe Bild 2a). Eine Zuweisung wie

```
D↑ := 99;
```

füllt den Puffer mit dem angegebenen Wert (Bild 2b). Natürlich erfolgen auch bei Files die üblichen Typüberprüfungen, so daß der Compiler die Zuweisung »D↑ :='K'« als fehlerhaft erkennt. Mit PUT(D) wird der Wert des Puffers in das File geschrieben und der Schreibzeiger eine Position weiter gesetzt. Jede Wiederholung der Schritte Wertzuweisung an den Puffer und Puffer schreiben erweiter das File um jeweils eine Komponente (Bild 2c):

```
D↑:= 43; PUT(D);
```

Es ist beim sequentiellen Schreiben nicht möglich, eine bereits geschriebene Komponente wieder zu korrigieren, da dazu der Schreibzeiger rückwärts bewegt werden müßte.

2. Sequentielles Lesen

Durch den Aufruf der Standardprozedur RESET mit der Filevariablen als Parameter wird die Puffervariable auf den Wert der ersten Komponente im File gesetzt (Bild 2d). Die Puffervariable kann man wie eine »normale« Variable in Ausdrücken verwenden und so das File Stück für Stück bearbeiten. Im Beispiel ergibt die Befehlsfolge

```
RESET(D); WRITE(D↑, D↑+5)
```

die Ausgabe »99 104«. Aufruf GET(D) setzt die Puffervariable wiederum eine Position weiter, so daß D↑ den Wert 43 enthält (Bild 2 e). Nach einem erneuten Aufruf von GET(D) steht das Lesefenster hinter der letzten belegten Komponente (Bild 2f) und die Puffervariable ist undefiniert. Um festzustellen, ob das Lesefenster über einem definierten Wert steht oder sich hinter dem Fileende befindet, dient die Standardfunktion

```
EOF(Filevariable)
```

So lieferte EOF(D) den booleschen Wert TRUE, falls D↑ sich in der letzten Komponente (wie in Bild 2f) befindet. Insbesondere ist EOF=TRUE, falls man ein leeres File mit RESET zum Lesen eröffnet. Da im Beispielprogramm in Listing 1 die Länge des Files unbekannt ist, wird eine Schleife der Form

```
WHILE NOT EOF(D) DO
```

verwendet. Übrigens ist per Definition EOF(D)=TRUE, falls auf das File D (nach RESET(D)) geschrieben wird.

Man kann ein File in beliebiger Reihenfolge mit RESET und REWRITE zum sequentiellen Lesen und Schreiben

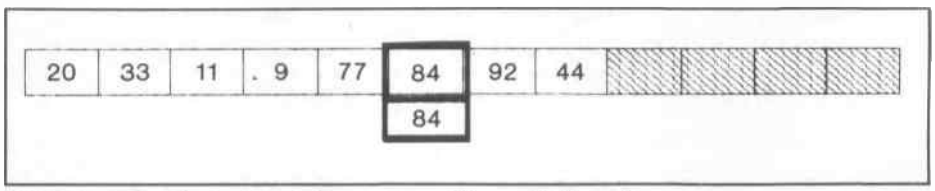


Bild 1. Schematische Darstellung eines Files mit Filepuffer

```
PROGRAM FILES (INPUT, OUTPUT);
  VAR ZAHLENDATEI: FILE OF INTEGER;
      X          : INTEGER;

BEGIN
  (* ZAHLEN VON DER TASTATUR AUF FILE SCHREIBEN: *)
  REWRITE(ZAHLENDATEI);          (* SEQUENTIELL SCHREIBEN *)
  REPEAT
    READLN(X);
    ZAHLENDATEI^:= X;            (* PUFFER MIT ZAHL FÜLLEN *)
    PUT(ZAHLENDATEI);            (* PUFFER SCHREIBEN       *)
  UNTIL X=0;

  (* ZAHLEN VOM FILE LESEN UND SCHREIBEN: *)
  RESET(ZAHLENDATEI);            (* SEQUENTIELL LESEN *)
  WHILE NOT EOF(ZAHLENDATEI) DO (* SOLANGE NICHT HINTER DEM DATEIENDE *
    BEGIN
      X:= ZAHLENDATEI^;          (* PUFFER AUSLESEN *)
      WRITELN(X);
      GET(ZAHLENDATEI);          (* PUFFER AUF NÄCHSTE ZAHL *)
    END;
END.
```

Listing 1. Grundlegende Fileoperationen

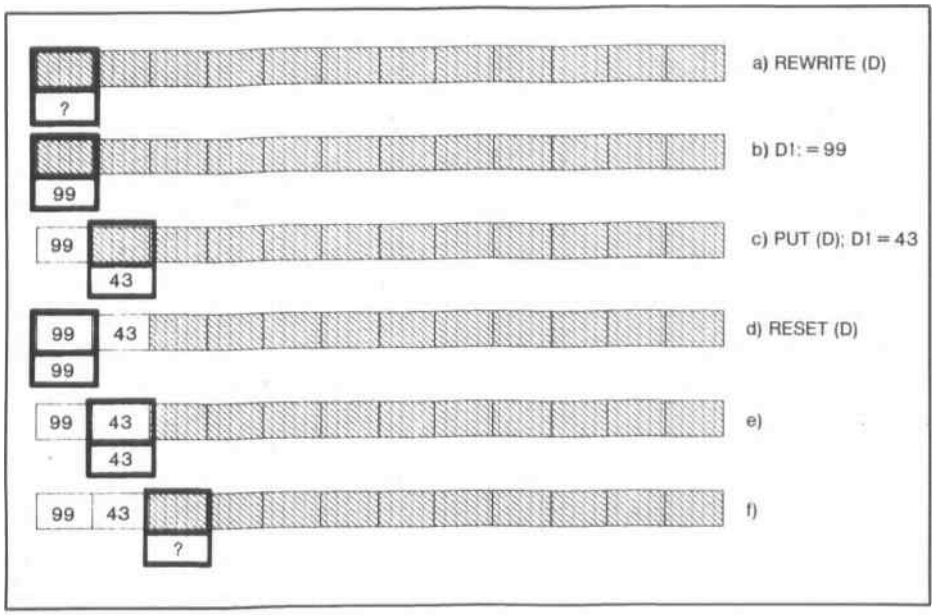


Bild 2. Operationen in einem File (a bis f)

```
PROGRAM MERGE(INPUT, OUTPUT);
(* 2 PHASEN 3-BAND NATÜRLICHES MISCHSORTIERTEN *)

TYPE ITEM = RECORD
              KEY: INTEGER;    (* ODER CHAR, REAL, ARRA(..) OF CHAR *)
              (* HIER KÖNNEN WEITERE FELDER STEHEN *)
            END;

     TAPE = FILE OF ITEM;

VAR  C   : TAPE;          (* DIESES FILE WIRD SORTIERT *)
     BUF: ITEM;           (* HILFSVARIABLE ZUR EINGABE *)


  PROCEDURE LIST (VAR F: TAPE);
  (* ZEIGE DEN INHALT VON F AN *)
    VAR X: ITEM;
  BEGIN
    RESET(F);
    WHILE NOT EOF(F) DO
      BEGIN
        X.KEY:= F^.KEY;
        GET(F);
        WRITE(X.KEY:4)
      END;
    WRITELN
  END; (* LIST *)

PROCEDURE NATURALMERGE;
(* SORITERE DAS FILE C (GLOBAL) IN MEHREREN DURCHLÄUFEN.      *)
(* DIE PROZEDUR BENUTZT ZWEI HILFSFILES A UND B.              *)
  VAR L  : INTEGER; (* ANZAHL DER LÄUFE AUF C                 *)
      EOR: BOOLEAN  (* END OF RUN, ENDE DES LAUFES            *)
      A,B: TAPE;    (* HILFSFILES                             *)

  PROCEDURE COPY(VAR X,Y: TAPE);
  (* KOPIERE RECORD VON X NACH Y, AKTUALISIERE EOR           *)
    VAR BUF: ITEM;
  BEGIN
    BUF.KEY:= X^.KEY; GET(X);
    Y^.KEY:= BUF.KEY; PUT(Y);
    IF EOF(X) THEN EOR:=TRUE
              ELSE EOR:=BUF.KEY>X^.KEY
  END; (* COPY *)

  PROCEDURE COPYRUN (VAR X,Y: TAPE);
  (* KOPIERE KOMPLETTEN LAUF VON X NACH Y *)
  BEGIN
    REPEAT COPY(X,Y) UNTIL EOR
  END; (* COPYRUN *)

  PROCEDURE DISTRIBUTE;
  (* KOPERE LÄUFE VON C ABWECHSELND AUF A UND B *)
  BEGIN
    REPEAT
```

Listing 2. Natürliches Mischsortieren mit Files

eröffnen, jedoch ist zu beachten, daß jeder Aufruf mit REWRITE alle im File existierenden Komponenten löscht. Möchte man ein bestehendes File am Ende erweitern, so muß man zunächst das gesamte File mit GET und PUT auf ein anderes File kopieren, an das man dann mit PUT die neuen Komponenten anfügt. Sollten Sie beim Test des Programmes in Listing 1 nicht das gewünschte Ergebnis erhalten, so liegt das wahrscheinlich daran, daß der Compiler für Ihren Computer zusätzliche Anweisungen zur Arbeit mit Files benötigt.

In Turbo-Pascal zum Beispiel verlangt eine Filevariable vor dem ersten RESET oder REWRITE mit der Prozedur ASSIGN einen

```
ASSIGN(D, 'DATEN.SEQ')
```

Außerdem sollten in Turbo-Pascal unter dem Betriebssystem PC-DOS alle Files (auch solche, von denen nur gelesen wurde) am Ende der Verarbeitung mit CLOSE geschlossen werden:

```
CLOSE(D)
```

Werfen Sie bei Schwierigkeiten also zunächst einen Blick in die Handbücher. Neben diesem zusätzlichen Aufwand bei der Behandlung sequentieller Dateien bieten viele Compiler die Möglichkeit, die Schreib- und Leseposition in einem File gezielt zu setzen. Mit

```
SEEK(D,400)
```

zeigt die Puffervariable auf die 400. Komponente im File. Arbeitet man mit einer Festplatte, so läßt sich ein File wie ein Array variabler Größe mit hundertfach langsamerer Zugriffszeit verwenden.

Zurück zum Pascal-Standard. Eine grundlegende Operation mit Files ist das Sortieren. Für das Sortieren von Arrays gibt es eine Vielzahl von Algorithmen, deren Verständlichkeit oft umgekehrt proportional zur Sortiergeschwindigkeit ist. Falls Sie Pascal und nicht das Sortieren auf externen Speichermedien lernen wollen, ist in Listing 2 ein Programm zum sogenannten »3-Band-, 2-Phasen-Mischsortieren« abgedruckt.

Natürlich kann man ein File dadurch sortieren, daß man es komplett in den Speicher lädt, dort sortiert und es anschließend geordnet zurückschreibt. Reicht jedoch (was häufig der Fall ist) der Arbeitsspeicher nicht aus, so hat man von jedem File in jedem Schritt nur die Komponente der Puffervariablen zur Verfügung. Um ein File umzusortieren, benötigt man also mindestens drei Files. Von einem File wird sequentiell gelesen, während die gelesenen Werte in möglichst geschickter Reihenfolge auf die beiden übrigen Files verteilt werden (Prozedur DISTRIBUTE). Anschließend werden die beiden erzeugten Files zum Lesen eröffnet und auf das

ursprüngliche Lesefile zurückgeschrieben (Prozedur MERGE), wobei jeweils zwei sortierte Teilsequenzen zu einer sortierten Sequenz auf dem Ausgangsfile verbunden werden.

Durch eine Wiederholung der zwei Phasen Verteilen und Mischen (dies ist der Fachausdruck für das Zusammenfassen zweier teilweise geordneter Files zu einem dritten, ebenfalls geordneten File) erhält man schließlich ein sortiertes File. Ein Beispiel soll das Verhalten des Programmes verdeutlichen:

C = (1 4 3 2 7 5 6 8) verteilt ergibt
A = (1 4 2 7)
B = (3 5 6 8) gemischt ergibt
C = (1 3 4 5 6 8 2 7) verteilt ergibt
A = (1 3 4 5 6 8)
B = (2 7) gemischt ergibt
C = (1 2 3 4 5 6 7 8)

Noch ein paar Worte zur Theorie: Eine geordnete Teilfolge innerhalb eines Files wird als Lauf (englisch run) bezeichnet. So besteht die Folge
1,4/3/2,7/5,6,8
aus Läufen. Demzufolge werden mit DISTRIBUTE Läufe von C abwechselnd nach A und B verteilt, und anschließend mit MERGE Läufe auf A und B zu Läufen auf C zusammengefaßt. Die Variable L zählt die Anzahl der Läufe auf C. Ist L=1, so muß also C aufsteigend geordnet sein.

Das Programm selbst zeigt praktisch alle Operationen, die mit Files in Pascal möglich sind. Zunächst wird ein Recordtyp ITEM vereinbart, der ein Schlüsselfeld KEY vom Typ INTEGER besitzt. Natürlich könnte an dieser Stelle auch ein anderer skalarer Typ oder auch der Typ ARRAY[1..N] OF CHAR stehen. Dann interpretiert der Compiler alle Vergleichsoperationen mit »=« und »><« korrekt.

Anschließend wird der eigentliche Filetyp TAPE deklariert. Jede Komponente der Files besteht also aus einem Record, so daß mit jedem Aufruf von PUT und GET ein kompletter Record zwischen Puffervariable und File übertragen wird.

C bezeichnet das zu sortierende File, das am Programmanfang in einer REPEAT-Schleife mit einer Folge ganzer Zahlen gefüllt wird. Die Prozedur LIST zeigt, daß man auch Files als Parameter übergeben kann. Dabei muß man jedoch Variablenmeter verwenden. In LIST wird zunächst das File F zum Lesen eröffnet und anschließend in der üblichen Weise mit einer WHILE-Schleife ausgelesen. Hinter dem Namen NATURALMERGE verbirgt sich die eigentliche Prozedur zum Sortieren. Den Beginn und das Ende des Anweisungsteils der Prozedur kennzeichnen Kommentare. Sicher ist Ihnen schon aufgefallen, daß man ein größeres File in Pascal am besten von hinten liest, da

```
      COPYRUN(C,A);
      IF NOT EOF(C) THEN COPYRUN(C,B);
    UNTIL EOF(C);
  END; (* DISTRIBUTE *)

  PROCEDURE MERGE;
  (* MISCHE FILE A UND B ZU FILE C. L (GLOBAL) ZÄHLT DIE LÄUFE *)

    PROCEDURE MERGERUN;
    (* MISCHE LÄUFE VON A UND B ZU LÄUFEN AUF C *)
    BEGIN
      REPEAT
        IF A^.KEY<B^.KEY THEN      (* KLEINEREN SCHLÜSSEL KOPIEREN *)
          BEGIN
            COPY(A,C);
            IF EOR THEN COPYRUN(B,C)
          END
        ELSE
          BEGIN
            COPY(B,C);
            IF EOR THEN COPYRUN(A,C)
          END
      UNTIL EOR;
    END; (* MERGERUN *)

    BEGIN (* MERGE *)
      L:=0;
      (* ZUNÄCHST FILES MISCHEN, BIS EINES ZU ENDE: *)
      REPEAT
        MERGERUN; L:=L+1
      UNTIL EOF(A) OR EOF(B);

      (* JETZT DEN REST VOM JEWEILIGEN FILE KOPIEREN *)
      WHILE NOT EOF(A) DO
        BEGIN
          COPYRUN(A,C);
          L:=L+1
        END;
      WHILE NOT EOF(B) DO
        BEGIN
          COPYRUN(B,C);
          L:=L+1
        END;
    END; (* MERGE *)

  BEGIN (* NATURALMERGE *)
    (* ABWECHSELND VERTEILEN UND MISCHEN,  BIS  NUR  EIN LAUF AUF C IST *)
    REPEAT
      REWRITE(A); REWRITE(B); RESET(C);  (* VON C NACH A UND B *)
      DISTRIBUTE;
      RESET(A); RESET(B); REWRITE(C)
      MERGE;
      LIST(C); (* NUR ZUR VERDEUTLICHUNG *)
    UNTIL L=1;
  END; (* NATURALMERGE *)

BEGIN (* HAUPTPROGRAMM *)
  WRITELN('SORTIEREN EINES SEQUENTIELLEN FILES:');
  WRITELN('EINGABEZAHLEN: (MIT 0 ABSCHLIESSEN)');

  REWRITE(C);
  READ(BUF.KEY);
  REPEAT
    C^.KEY:=BUF.KEY; PUT(C);       (* AUF C SCHREIBEN *)
    READ(BUF.KEY)
  UNTIL BUF.KEY=0;

  LIST(C);          (* ZUR KONTROLLE EINGABEZAHLEN ANZEIGEN *)
  NATURALMERGE;     (* SORTIERE C *)
END.
```

Listing 2. Natürliches Mischsortieren mit Files (Schluß)

durch die Schachtelung der Prozeduren die Anweisungsteile des Hauptprogrammes und der wichtigen Prozeduren eben nach »hinten« wandern.

NATURALMERGE selbst besteht, wie bereits besprochen, im wechselseitigen Aufruf der beiden Phasen DISTRIBUTE und MERGE, bis C nur noch einen Lauf umfaßt. In der Hierarchie der Prozeduren steht die Prozedur COPY am weitesten unten. Sie kopiert eine Komponente vom File X zum File Y. Um das Ende eines Laufes zu erkennen, findet hier die boolesche Variable EOR (end of run, Ende des Laufs) Anwendung. So ist nach der Zuweisung

EOR:= BUF.KEY >X↑.KEY

die Variable EOR = TRUE, falls das Feld KEY im Record BUF größer als das entsprechende Feld in der Puffervariablen des Files X ist. Da nach jedem Durchlauf die bisher erreichte Sortierung auf dem File C angezeigt wird, sehen Sie mit den folgenden Testdaten die Funktionsweise des Programmes am deutlichsten:

8,7,6,5,4,3,2,1,0

Als nächstes wenden wir uns einem speziellen Typ von Files zu. Während die Files, die mit PUT und GET erzeugt werden, die Daten Byte für Byte wie im Speicher darstellen, ist es oft sinnvoll,

Ein- und Ausgaben zu verwenden, die ein Mensch verstehen kann. Konkret gesprochen stellt man sich unter der Zahl 12345 die Zeichenfolge »12345« und nicht etwa zwei Byte ($30 und $39 hexadezimal) vor.

Da Files, die aus Zeichenfolgen bestehen, eine große Rolle spielen, existiert in Pascal ein vordefinierter Typenbezeichner

```
TYPE TEXT = FILE OF CHAR;
```

Mit der Deklaration

```
VAR F: TEXT;
```

können neben den Operationen RESET und REWRITE auf das Textfile F auch formatiert Werte ein- und ausgegeben werden. Das Prinzip besteht darin, den Prozeduren READ, READLN, WRITE und WRITELN als ersten Parameter eine Filevariable zu übergeben, von der die Eingabe kommt oder auf welche die Ausgabe erfolgen soll:

(1) WRITE(F,AUSDRUCK : n)

Mit diesem Prozeduraufruf wird der Ausdruck vom Typ INTEGER, REAL, CHAR oder BOOLEAN auf das File F in ein Feld mit einer Mindestlänge von n Zeichen geschrieben. Ist die Länge des auszugebenden Wertes größer als n, so wird n ignoriert:

```
WRITE(123456 : 10); WRITE('*': 3)
```

erzeugt also folgende Ausgabe:

```
____123456____*
```

Ist der Ausdruck vom Typ REAL, so sind nach dem Doppelpunkt zwei Formatierungsangaben möglich:

```
WRITE(F,reeller Ausdruck: n : m)
```

druckt die reale Zahl nicht in Gleitkommadarstellung (zum Beispiel 1.00000E+2), sondern in Festkommadarstellung. Dabei gibt n wieder die Mindestgröße des Feldes an, in das die Zahl ausgegeben wird, wohingegen m die Anzahl der angezeigten Nachkommastellen wiedergibt:

```
WRITELN(1.2345 : 10: 5)
```

ergibt die Ausgabe

```
___1.23450
```

(2) WRITELN(F)

Wichtig ist die Tatsache, daß sich die Zeichen auf einem Textfile in Zeilen gliedern. Mit dem Prozeduraufruf WRITELN(F) wird auf dem Textfile F ein Zeilenende erzeugt. Technisch bewirkt dies die Ausgabe von ein oder zwei Steuerzeichen (CR, eventuell LF, carriage return und line feed, also Wagenrücklauf und Zeilenvorschub) auf das File F.

Eine Folge von Ausgaben, wie

```
WRITE(F,A); WRITE(F,B); WRITE(F,C);
```

kann immer zusammengefaßt werden zu

```
WRITE(F,A,B,C)
```

Außerdem entspricht der Aufruf

```
WRITELN(F,A,B,...)
```

der Befehlsfolge

```
WRITE(F,A,B,...);WRITELN(F)
```

Somit erzeugt also die Ausgabe mit WRITELN immer einen Zeilenwechsel am Ende der Ausgabe. Natürlich muß vor allen Schreiboperationen das File F mit REWRITE(F) zum Schreiben eröffnet worden sein. Dabei werden wieder alle Zeichen, die eventuell zuvor in F existierten, gelöscht. Beabsichtigt man eine Ausgabe auf dem Standard-Ausgabegerät (dem Bildschirm), so bietet sich als Filevariable die vordeklarierte Variable OUTPUT an. Diese kann man sich mit der Deklaration

```
VAR OUTPUT: TEXT;
```

vereinbart denken. In Wirklichkeit wird man jedoch in diesem Fall den Fileparameter bei WRITE nicht angeben, und erhält somit die vereinfachte Syntax, die Sie bereits am Anfang des Artikels kennengelernt hatten:

```
WRITE(OUTPUT, 'Hallo Bildschirm')
```

läßt sich also ersetzen durch

```
WRITE('Hallo Bildschirm').
```

(3) READ(F, Variable)

Vom Textfile F werden Zeichenfolgen eingelesen und als Werte des Typs der Variablen interpretiert. Am besten kann man diese Umwandlung von Zeichenfolgen auf dem File in Werte, zum Beispiel des Typs INTEGER, an Beispielen verdeutlichen:

```
VAR I: INTEGER;
    R: REAL;
    C: CHAR;
```

```
PROGRAM TEXTFILES (INPUT,OUTPUT);

  PROCEDURE FORMAT(VAR EINGABE, AUSGABE: TEXT; RECHTS: INTEGER);
  (* FORMATIERE VON EINGABE NACH AUSGABE AUF RECHTEN RAND IN DER SPALTE  *)
  (* RECHTS. LEERE ZEILEN, ODER ZEILEN MIT NUR EINEM WORT WERDEN NICHT   *)
  (* VERÄNDERT.                                                          *)

  CONST SPACE     = ' ';
  TYPE  LINEINDEX = 1..136;
  VAR   LINE      : ARRAY [LINEINDEX] OF CHAR;
        BIS,I: LINEINDEX;
        ZEILE: INTEGER; (* LFD. ZEILENNUMMER                              *)
        S    : INTEGER; (* ANZAHL DER EINZUFÜGENDEN LEERZEICHEN IN ZEILE *)
        P,Q  : INTEGER; (* ANZAHL DER LEERZEICHEN NACH JEDEM WORT         *)
        T    : INTEGER; (* INDEX DES WORTES, NACH DEM ZUM 1. MAL Q LEER-  *)
                        (* ZEICHEN EINGEFÜGT WERDEN SOLLEN                *)
        K,N  : INTEGER; (* ANZAHL DER WORTE IN DER ZEILE                  *)

    PROCEDURE READLINE(VAR LASTCOLUMN: INTEGER; VAR WORDS: INTEGER);
    (* KOMPLETTE ZEILE EINLESEN UND IN LINE (GLOBAL) SPEICHERN    *)
    (* IN LASTCOLUMN STEHT DAS LETZTE ZEICHEN UNGLEICH SPACE IM   *)
    (* TEXT. HINTER DEM TEXT EIN SPACE. WORD ZÄHLT DIE DURCH      *)
    (* LEERZEICHEN GETRENNTEN WORTE.                              *)

      VAR  WASSPACE, ISSPACE: BOOLEAN;
           I                : LINEINDEX;
           CH               : CHAR;
    BEGIN
      I:=1; WASSPACE:= TRUE; WORDS:= 0; LASTCOLUMN:=0;
      WHILE NOT EOLN(EINGABE) DO
        BEGIN
          READ(EINGABE,CH); LINE[I]:= CH;
          ISSPACE:= CH=SPACE;
          IF NOT ISSPACE THEN
            BEGIN
              LASTCOLUMN:= I;
              IF WASSPACE THEN WORDS:= WORDS +1;   (* WORTANFANG *)
            END;
          WASSPACE:= ISSPACE; I:= I+1
        END;
      LINE[LASTCOLUMN+1]:= SPACE; READLN(EINGABE);
    END; (* READLINE *)

    PROCEDURE COPYWORD;
    (* AB DER MOMENTANEN POSITION IN LINE WORT MIT VORLAUFENDEN SPACES *)
    (* BIS ZUM NÄCHSTEN SPACE AUSGEBEN                                 *)
    BEGIN
      WHILE LINE[I]=SPACE DO
        BEGIN WRITE(AUSGABE,SPACE); I:= I+1 END;
      REPEAT
        WRITE(AUSGABE,LINE[I]); I:=I+1
      UNTIL LINE[I]=SPACE;
    END; (* COPYWORD *)

    PROCEDURE INSERTSPACES(N: INTEGER);
    (* N LEERZEICHEN AUSGEBEN *)
    BEGIN
      IF N>0 THEN WRITE (AUSGABE, SPACE : N)
    END; (* INSERTSPACES *)

    BEGIN (* FORMAT *)
      RESET(EINGABE); REWRITE(AUSGABE);
      ZEILE:= 0;

      WHILE NOT EOF(EINGABE) DO
        BEGIN
          READLINE(BIS,N); ZEILE:= ZEILE + 1;
          S:= RECHTS-BIS;    (* ANZAHL DER FEHLENDEN LEERZEICHEN *)
```

Listing 3. Kopieren und Formatieren eines Textfiles

Befindet sich auf dem File F folgende Zeichenfolge:

```
1234        34.55X
```

so bewirkt der Prozeduraufruf READ (F,I,R,C) folgende Zuweisungen:

```
I:=1234; R:=34.55; C:='X'
```

Ist der Parameter bei READ eine ganze oder reelle Zahl, so werden zunächst Leerzeichen und Zeilenwechsel ignoriert. Anschließend wird eine Ziffernfolge eingelesen und der entsprechende Wert der Variablen zugewiesen. Eine nachfolgende READ-Operation verarbeitet das Zeichen, das direkt hinter der gelesenen Zahl beginnt. Daher besitzt die Variable C im obigen Beispiel den Wert »X«.

(4) READLN(F)

Vom File F werden solange Zeichen eingelesen, bis ein Zeilenende erkannt wurde. Der nächste Aufruf der Prozedur READ mit dem File F liest das erste Zeichen der folgenden Zeile. Es gibt ebenfalls ein Standardfile zur Eingabe, das folgendermaßen vordeklariert ist:

```
VAR INPUT: TEXT;
```

Wiederum kann man Eingaben von der Tastatur als Standard-Eingabe auch ohne Angabe einer Filevariablen erreichen. READ(INPUT,I,R,C) entspricht READ(I,R,C) und liest drei Werte von der Tastatur.

(5) EOLN(F)

Die Standardfunktion EOLN, angewandt auf ein Textfile F, liefert einen booleschen Wert. Er ist genau dann TRUE, falls bei der letzten Eingabe vom File F das Zeilenende erreicht wurde. Bitte beachten Sie, daß Sie beim Einlesen nie die oben genannten Steuerzeichen (Zeilenwechsel oder Wagenrücklauf) erhalten, da diese vom Pascal-Laufzeitsystem automatisch in Leerzeichen (blanks) umgewandelt werden.

Nach diesen zugegebenermaßen recht detaillierten Ausführungen über Textfiles in Pascal sollen Sie zur Belohnung auch ein wirklich sinnvolles Programm kennenlernen. Es kopiert einen Text von einem Textfile zu einem anderen und formatiert dabei Zeilen rechtsbündig (Listing 3).

Die Prozedure FORMAT wird mit drei Parametern aufgerufen. EINGABE liest den zu formatierenden Text und schreibt ihn auf AUSGABE. RECHTS enthält die Spalte, in der jede Zeile enden soll. Für einen Testlauf können Sie RECHTS beispielsweise auf 40 Zeichen setzen.

Nachdem EINGABE und AUSGABE korrekt zum Lesen und Schreiben eröffnet wurden, wird jeweils eine Zeile mit READLINE eingelesen, entschieden, ob die Zeile formatiert werden muß und schließlich die formatierte Zeile ausgegeben. Beim Einlesen (READLINE) wird der Text in einen Zeilenpuffer LINE geschrieben. Alle Indizes in diesem

```
        I:=1;
        IF (N<=1) OR (S<=0) THEN (* 1:1 KOPIEREN *)
          WHILE I<=BIS DO
            BEGIN WRITE(AUSGABE, LINE[I]); I:= I+1 END
        ELSE
          BEGIN (* BERECHE ANZAHL DER LEERZEICHEN UND VERTEILUNG: *)
            IF ODD(ZEILE) THEN
              BEGIN                          (* VON LINKS EINFÜGEN *)
                P:= S DIV (N-1);
                Q:= P+1;
                T:= P * (N-1) + N - S
              END
            ELSE
              BEGIN                          (* VON RECHTS EINFÜGEN *)
                Q:= S DIV (N-1);
                P:= Q+1;
                T:= S + 1 - Q * (N-1)
              END;
            FOR K:= 1 TO N-1 DO              (* WORTE KOPIEREN UND ERWEITERN *)
              BEGIN
                COPYWORD;
                IF K>=T THEN INSERTSPACES(Q)
                        ELSE INSERTSPACES(P)
              END;
            COPYWORD;                        (* LETZTES WORT *)
          END; (* IF *)
        WRITELN(AUSGABE);
      END; (* WHILE NOT EOF... *)
  END; (* FORMAT *)

BEGIN
  FORMAT(INPUT, OUTPUT, 40);
END.
```

Listing 3. Kopieren und Formatieren eines Textfiles (Schluß)

ARRAY von Zeichen sind als Ausschnittstypen

```
TYPE LINEINDEX = 1..136
```

deklariert. Außerdem bestimmt READLINE die Anzahl der Worte in der Zeile und die letzte Spalte, in der ein Buchstabe stand. Diese Werte kommen als Variablenparameter zurück. Anschließend wird die Anzahl der Leerzeichen bestimmt, die in die Zeile einzufügen sind, damit das letzte Zeichen in Spalte RECHTS erscheint. Enthält die Zeile nur ein Wort oder fehlen keine Leerzeichen, so kann der Zeilenpuffer LINE ohne Änderung ausgegeben werden.

Ansonsten beginnt die eigentliche Formatierung. Dabei sollten Sie vermeiden, daß sich die Worte alle am rechten und linken Rand sammeln. Daher werden in Zeilen mit ungerader Zeilennummer (»ODD(ZEILE)=TRUE«) Leerzeichen von links eingefügt, sonst jedoch von rechts.

Ein Beispiel: Steht in einer Zeile 11 Mal das Zeichen A mit einem Zwischenraum und ist RECHTS=40, so müssen 18 Leerzeichen auf 10 Wortzwischenräume verteilt werden. Ein Teil der Zwischenräume wird also um ein Leerzeichen, der Rest um zwei Leerzeichen erweitert. Im Programm übernehmen diese Verteilung die Variablen P, Q und T. Die ersten Worte werden mit P Leerstellen am Ende erweitert, während ab dem T-ten Wort Q Leerstellen angefügt werden. In der FOR-Schleife enthält also K die Nummer des gedruckten Wortes.

Nach diesen Ausführungen kennen Sie die Logik des Programmes, so daß Sie noch ein paar technische Details zum Thema Textfiles aufnehmen können: Zunächst erkennen Sie am Aufruf »FORMAT(INPUT,OUTPUT,40)« den Nutzen der vordefinierten Textfiles INPUT und OUTPUT, die ja auch im Programmkopf angegeben werden. Mit ihnen kann man die Tastatur und den Bildschirm wie ein normales File ansprechen. Bei der Prozedur READLINE wird die Funktion EOLN(Eingabe) zum Erkennen des Zeilenendes benutzt. Damit nach der Bearbeitung der ersten Zeile beim nächsten Aufruf die folgende Zeile weiterbearbeitet wird, muß am Ende der Aufruf READLN(EINGABE) stehen. Die Prozedur INSERTSPACES verwendet die Angabe eines Formatierungsparameters nach dem Doppelpunkt, um eine definierte Anzahl von Leerzeichen zu drucken:

```
WRITE(AUSGABE, ' ': N)
```

druckt ein Leerzeichen rechtsbündig in einem Feld der Größe N. Somit werden insgesamt N Leerzeichen ausgegeben. Schließlich erfolgt nach der Ausgabe jeder Zeile der Aufruf WRITELN(AUSGABE). Damit wird auf dem File AUSGABE ein Zeilenwechsel erzeugt, um die Zeilenstruktur des Files EINGABE zu erhalten.

Damit ist die Diskussion des Datentyps File und speziell der Textfiles beendet. Als Übung können Sie versuchen, das Programm in Listing 3 so zu verändern, daß es jede Zeile zentriert druckt. Sie können also die Prozedur READLINE wieder verwenden, jedoch dafür sorgen, daß die fehlenden Leerstellen zur Hälfte vor dem ersten Wort gedruckt werden.

(Florian Matthes/ev)

Von Zeigern, Listen und Graphen (Pascal, Teil 4)

Der letzte Beitrag unserer Einführung in die Programmiersprache Pascal beschäftigt sich mit einem nicht ganz einfachen, aber sehr interessanten Thema: den dynamischen Datenstrukturen.

Die Beschreibung der dynamischen Datenstrukturen steht grundsätzlich am Ende jeder Einführung in Pascal. Dies liegt einerseits daran, daß es sich dabei um ein besonders leistungsfähiges Sprachelement handelt, andererseits möchte man den Anfänger erst zum Schluß mit einem völlig neuen Verfahren zur Behandlung von Variablen konfrontieren. Haben Sie also in den vorhergehenden Artikeln Ihre ersten Schritte in Pascal gemeistert, dürfen Sie an dieser Stelle nicht frustriert zur Überzeugung gelangen, daß Pascal viel zu kompliziert und unverständlich ist. Vielmehr sollten Sie, nachdem Sie das bisher Gelernte in eigenen Programmen verwendet haben, mit diesen Erfahrungen den letzten Teil zur Fortbildung nutzen. Sicherlich gibt es auch einige Leser, die bisher zwar in Pascal programmiert haben, dabei jedoch einen weiten Bogen um Pointervariablen geschlagen haben. Für diese beginnt jetzt wohl der eigentlich interessante Teil der Artikelserie.

Alle bisher behandelten Datentypen und Variablen waren statisch. Am Beginn jedes Blockes wurden die lokalen Variablen angelegt und waren über ihren Namen veränderlich, bis der Block wieder verlassen und das Ende der Gültigkeit der Variablen erreicht wurde. Diese Variablenverwaltung hat zur Folge, daß bereits zur Übersetzungszeit der Compiler eine Speicherplatzverwaltung durchführen kann. Außerdem entspricht jedem Variablennamen ein (eventuell zusammengesetzter) Wert, jedoch gibt es auch gravierende Nachteile. Jedes Array besitzt eine feste Größe (es gibt keinen variablen DIM-Befehl wie zum Beispiel in Basic), so daß man oft entweder ein zu kleines Array definiert oder unnötig Speicherplatz verschwendet.

Man möchte also zur Laufzeit des Programms dynamisch entscheiden, ob Speicherplatz für eine Variable anzulegen ist oder ob eine bestehende Variable gelöscht werden soll. Außerdem möchte man auf diese dynamisch erzeugten Variablen gezielt zugreifen. Andererseits will man nicht auf die Typüberprüfungen des Compilers verzichten. Die Lösung des Dilemmas besteht darin, Variablen nicht mehr durch Namen, sondern durch Zeiger zu identifizieren.

Betrachten wir ein konkretes Beispiel. Es soll eine Kundenliste gebildet werden. Von jedem Kunden wird Name und Kundennummer gespeichert. Da die Anzahl der Kunden (in der Zukunft) unbekannt ist, scheidet ein Array von Kundenrecords als Datenstruktur aus. Statt die Daten auf einem langsamen externen Datenspeicher als File abzulegen, wird eine Liste mit Zeigern gebildet (Bild 1). Bildlich gesprochen entspricht jeder Record einem Kasten, der alle Daten eines Kunden enthält. Um nun einen Record im Programm anzusprechen, benutzt man keinen Variablennamen, sondern einen Zeiger auf diesen Kasten.

```
TYPE KUNDENZEIGER = ↑ KUNDE;
     KUNDE = RECORD
               NAME: ARRAY[1..10]
               OF CHAR;
               KNUMMER: INTEGER;
               NAECHSTER: KUNDEN-
               ZEIGER;
             END;
VAR KUNDE1, KUNDENEU, LETZTER:
KUNDENZEIGER;
```

In Bild 1 zeigt also der Zeiger KUNDE1 auf den ersten Kundenrecord. Von dort führt ein weiterer Zeiger zum nächsten Kundenrecord und so weiter. Jeder Zeiger (pointer) ist an einen Typ gebunden. So kann also eine Variable vom Typ KUNDENZEIGER nur auf einen Record vom Typ KUNDE weisen. Die Liste in Bild 1 ist also über einen Zeiger (KUNDE1) zugänglich. Indem man den Zeigern von Record zu Record folgt, kann man nacheinander jeden Record in der Liste adressieren.

Die obige Variablendeklaration definierte jeweils nur Speicherplatz für Zeiger auf Kundenrecords, jedoch keine Records selbst. Dies geschieht erst während der Programmausführung mit der Standardprozedur NEW:

```
NEW(KUNDE1)
```

Damit reserviert man irgendwo im Hauptspeicher des Rechners Speicherplatz für eine Variable des Typs, auf den die Pointervariable KUNDE1 zeigt. Um diesen neu erzeugten Kundenrecord zu adressieren, wird gleichzeitig dem Zeiger KUNDE1 die Adresse dieses Records zugewiesen (Bild 2a). Jetzt kann man der so erzeugten Variablen Werte übergeben:

```
KUNDE1 ↑ .NAME:= 'MAIER      ';
KUNDE1 ↑ .KNUMMER:= 100;
```

Während KUNDE1 eine Zeigervariable (vom Typ KUNDENZEIGER) ist, bezeichnet KUNDE1↑ eine Variable des Typs KUNDE. Indem man also den Pfeil hinter eine Zeigervariable stellt, erhält man die dynamische Variable, auf die die Pointervariable zeigt. Man bezeichnet deshalb den Pfeil auch als »Dereferenzier-Operator«.

Da KUNDE1↑ eine (dynamische) Recordvariable ist, folgen nach einem Punkt wie üblich die Feldnamen des Records. Damit erhält man den Zustand aus Bild 2b. Um einen weiteren Kunden in die Liste aufzunehmen, ist zunächst wieder Speicherplatz zu reservieren.

```
NEW(KUNDENEU)
```

Wie oben kann man jetzt diesen Record mit Werten füllen (2c):

```
KUNDENEU↑ .NAME:= 'MÜLLER    ';
KUNDENEU↑ .KNUMMER:= 200;
```

Schließlich soll KUNDENEU als Nachfolger von KUNDE1 eingetragen werden. Hierzu wird im Feld NAECH-

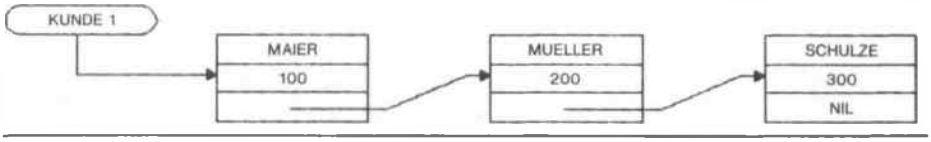


Bild 1. Eine Liste mit Zeigern

STER des Records, der durch KUNDE1 referiert wird, der Zeiger KUNDENEU eingetragen (Bild 2d).

```
KUNDE1↑ .NAECHSTER:= KUNDENEU
```

Um den letzten Record hinter KUNDENEU einzufügen, kann man folgende Anweisungsfolge verwenden:

```
NEW(LETZTER);
LETZTER↑ .NAME:='SCHULZE    ';
LETZTER↑ .KNUMMER:=300;
KUNDENEU↑ .NAECHSTER:= LETZTER
```

Damit ergibt sich eine Liste wie in Bild 2e. Wie erkennt man nun aber das Ende der Liste? Man muß wissen, ob das Feld NAECHSTER einen gültigen Zeiger enthält. Um anzuzeigen, daß ein Zeiger auf keine dynamische Variable weist, verwendet man den Wert NIL. Diese Konstante darf jeder Zeigervariablen zugewiesen werden. Mit

```
LETZTER↑.NAECHSTER:=NIL
```

gibt man also an, daß nach LETZTER↑ in der Liste kein Record mehr folgt.

Bisher programmierten wir alle Einfügungen in die Liste »zu Fuß«. Ein komplettes Programm zur Verwaltung einer Kundenliste zeigt Listing 1. Es speichert die Kunden in alphabetischer Reihenfolge. Zusätzlich existieren am Anfang und Ende der Liste je ein leerer Record. Damit ergibt sich eine Listenstruktur wie in Bild 3. Die Zeiger KOPF und ENDE weisen immer auf die beiden leeren Records. Im folgenden werden alle Funktionen des Programms anhand von Abbildungen erklärt.

Am einfachsten ist die Ausgabe der Tabelle (Bild 4a), siehe Prozedur TABELLE in Listing 1: Man durchläuft mit dem Zeiger Z die gesamte Liste und zeigt den jeweiligen Kundenrecord an. Mit

```
Z:= KOPF↑ .NAECHSTER
```

wird zunächst der leere (schraffierte) Record am Listenanfang übersprungen. Solange der Zeiger Z nicht mit dem Zeiger ENDE übereinstimmt, wird der Record Z↑ (nicht der Zeiger Z!) angezeigt. »Z:= Z↑ .NAECHSTER« führt schließlich von jedem Record zu seinem Nachfolger in der Liste.

Die Prozedur EINGABE liest zunächst von der Tastatur einen Namen ein. Dann wird durch den Aufruf der Prozedur VORHANDEN geprüft, ob dieser Name bereits in der Liste steht. Ist dies der Fall, so endet die Eingabe. Ansonsten wird dann ein neuer Record NEU geschaffen und mit Namen und Kundennummer gefüllt (Bild 4b). Da die Liste alphabetisch sortiert bleiben soll, muß NEU direkt hinter dem alphabetischen Vorgänger eingehängt werden. Daher liefert die Prozedur VORHANDEN einen Zeiger VOR, der in jedem Fall auf den Vorgänger in der Liste zeigt. Die Einfügung selbst geschieht dann in zwei Schritten. Zunächst wird der Zeiger NAECHSTER im neuen Record auf

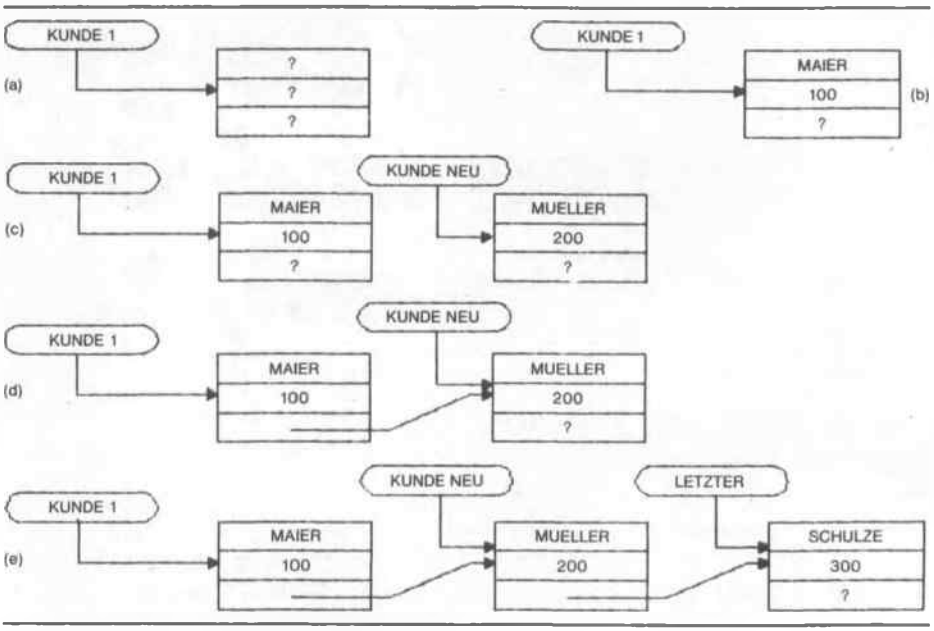


Bild 2. Operationen in der Liste (siehe Text)

```
PROGRAM KUNDENLISTE (INPUT, OUTPUT);
(* BEISPIEL FÜR DIE VERWALTUNG EINER LISTE MIT DYNAMISCHEN VARIABLEN *)
(* DIE DATEN VOM TYP KUNDE WERDEN STÄNDIG SORTIERT IN EINER EINFACH  *)
(* VERKETTETEN LISTE GESPEICHERT. JEDER RECORD BESITZT DAZU EINEN    *)
(* ZEIGER AUF DEN ALPHABETISCHEN NACHFOLGER. UM DAS EINFÜGEN UND     *)
(* LÖSCHEN EINFACH ZU GESTALTEN, BESITZT DIE LISTE JE EINEN LEEREN   *)
(* RECORD AM ANFANG UND ENDE.                                        *)

  CONST LEN = 10;              (* LÄNGE EINES NAMENS IN ZEICHEN      *)

  TYPE  STRING = ARRAY (1..LEN) OF CHAR;
        KUNDENZEIGER = ^ KUNDE;
        KUNDE  = RECORD
                   NAME      : STRING;
                   KNUMMER   : INTEGER;
                   NAECHSTER: KUNDENZEIGER;
                 END;
VAR  KOPF: KUNDENZEIGER; (* ZEIGER AUF DEN ERSTEN RECORD IN DER LISTE *)
     ENDE: KUNDENZEIGER; (* ZEIGER AUF DEN LETZTEN RECORD             *)
     CH  : CHAR;         (* BENUTZEREINGABE                           *)

PROCEDURE READSTRING (VAR S: STRING);
(* STRING MIT LEN ZEICHEN VON DER TASTATUR LESEN *)
  VAR I: INTEGER;
      C: CHAR;
BEGIN
  REPEAT READ(C) UNTIL C<>' '; (* VORLAUFENDE LEERZEICHEN IGNORIEREN  *)
  I:=1;

  REPEAT                       (* LEN ZEICHEN ODER BIS ZUM ZEILENENDE *)
    S[I]:= C; I:= I+1;
    READ(C)
  UNTIL (I>LEN) OR EOLN;

  WHILE I<LEN DO               (* S MIT LEERZEICHEN AUF DIE VOLLE     *)
    BEGIN                      (* LÄNGE ERWEITERN                     *)
      S[I]:= ' '; I:= I+1;
    END;
  READLN
END; (* READSTRING *)
FUNCTION VORHANDEN (S: STRING; VAR Z: KUNDENZEIGER): BOOLEAN;
(* SUCHT DEN NAMEN S IN DER LISTE. ERGEBNIS = TRUE, FALLS S GEFUNDEN *)
(* WURDE. Z ZEIGT BEI DER RÜCKKEHR IMMER AUF DIE POSITION DES ALPA-  *)
(* BETISCHEN VORGÄNGERS.                                             *)

    VAR Z1: KUNDENZEIGER;  (* Z1 STEHT IMMER EINEN RECORD WEITER ALS *)
                           (* DER ZEIGER Z                           *)
BEGIN
  Z:= KOPF; Z1:= KOPF^.NAECHSTER; (* Z1 ZEIGT AUF ERSTEN GÜLTIGEN     *)
                                  (* RECORD IN DER LISTE             *)
  ENDE^.NAME:=S;                  (* MARKE IM LEZTEN RECORD DER LISTE*)

  WHILE Z1^.NAME<S DO             (* WANDERE MIT Z1 DURCH DIE LISTE, *)
    BEGIN                         (* BIS POSITION VON S ERREICHT     *)
      Z:= Z1; Z1:= Z1^.NAECHSTER
    END;
  (* GEFUNDEN, FALLS NAMEN GLEICH UND NICHT MARKE ERREICHT           *)
  VORHANDEN:= (Z1^.NAME=S) AND (Z1<>ENDE);
END; (* VORHANDEN *)

PROCEDURE DRUCKE (Z: KUNDENZEIGER);
```

Listing 1. Kundenverwaltung mit Listenstruktur

```
  BEGIN
    WITH Z^ DO
      WRITELN('NAME:', NAME: LEN+2, '   NUMMER:', KNUMMER : 5)
  END; (* DRUCKE *)

  PROCEDURE EINGABE;
  (* EINFÜGEN EINES NEUEN KUNDENRECORDS AN DER KORREKTEN POSITION IN   *)
  (* DER KUNDENLISTE ZWISCHEN VON UND BIS                              *)
    VAR N   : STRING;          (* NEUER NAME                          *)
        NEU: KUNDENZEIGER; (* ZEIGER AUF DEN NEUEN RECORD             *)
        VOR: KUNDENZEIGER; (* ZEIGER AUF DEN VORGÄNGER IN DER LISTE   *)

  BEGIN
    WRITE('NAME:'); READSTRING(N);
    IF VORHANDEN (N,VOR) THEN     (* VOR WIRD HIER GESETZT!            *)
      WRITELN(N, ' IST BEREITS KUNDE!')
    ELSE
      BEGIN
        NEW (NEU);                (* NEUEN RECORD BESORGEN             *)
        WRITE ('KUNDENNUMMER:');
        READLN (NEU^.KNUMMER);    (* UND MIT WERTEN BELEGEN:           *)
        NEU^.NAME:= N;

        (* NEU^  HINTER VOR^  EINFÜGEN: *)
        NEU^.NAECHSTER:= VOR^.NAECHSTER;
        VOR^.NAECHSTER:= NEU
      END;
  END; (* EINGABE *)

  PROCEDURE AUSGABE;
  (* AUSGABE EINES KUNDENRECORDS                                       *)
    VAR N   : STRING;
        VOR: KUNDENZEIGER;  (* ZEIGER AUF ALPHABETISCHEN VORGÄNGER     *)
  BEGIN
    WRITE ('NAME:');
    READSTRING(N);
    IF VORHANDEN (N,VOR) THEN   (* VOR WIRD AUF DEN VORGÄNGER GESETZT! *)
      DRUCKE (VOR^.NAECHSTER)
    ELSE
      WRITELN (N, ' NICHT ALS KUNDE GESPEICHERT!');
  END; (* AUSGABE *)

  PROCEDURE LOESCHEN;
  (* LOESCHEN EINES KUNDENRECORDS IN DER LISTE                         *)
    VAR N   : STRING;
        VOR: KUNDENZEIGER;  (* WIEDERUM ZEIGER AUF DEN VORGÄNGER IN   *)
                            (* DER LISTE DER KUNDEN                   *)

  BEGIN
    WRITE('NAME:'); READSTRING(N);
    IF VORHANDEN(N,VOR) THEN  (* VOR WIRD AUF DEN VORGÄNGER GESETZT    *)
      BEGIN
        WRITELN ('GELÖSCHT WURDE:');
        DRUCKE (VOR^.NAECHSTER); (* ZUR SICHERHEIT ANZEIGEN            *)

        (* JETZT DEN NACHFOLGER VON VOR^ AUS DER LISTE ENTFERNEN:      *)
        VOR^.NAECHSTER:= VOR^.NAECHSTER^.NAECHSTER;
      END
    ELSE
      WRITELN (N, ' NICHT ALS KUNDE GESPEICHERT!');
  END; (* LOESCHEN *)

  PROCEDURE TABELLE;
  (* DRUCKE EINE ALPHABETISCHE AUFLISTUNG ALLER KUNDEN *)
    VAR Z: KUNDENZEIGER;
  BEGIN
    Z:= KOPF^.NAECHSTER;        (* Z AUF DEN ERSTEN BELEGTEN RECORD    *)
    WHILE Z<>ENDE DO            (* SOLANGE NICHT DER LETZTE (LEERE)    *)
      BEGIN                     (* RECORD ERREICHT WIRD:               *)
        DRUCKE(Z);              (* ANZEIGE Z^                          *)
        Z:= Z^.NAECHSTER;       (* ZUM NÄCHSTEN KUNDEN                 *)
      END;
  END; (* TABELLE *)

BEGIN (* HAUPTPROGRAMM *)
  NEW(KOPF); NEW(ENDE);         (* ERSTEN UND LETZTEN RECORD BILDEN    *)
  KOPF^.NAECHSTER:= ENDE;       (* ENDE DIREKT NACH DEM KOPF, ALSO IST *)
                                (* DIE LISTE LEER                      *)
  REPEAT
    WRITELN('E INGABE     L OESCHEN');
    WRITELN('A USGABE     T ABELLE');
    WRITELN('      X    BEENDEN');
    READLN(CH);
    IF CH IN ['E','A','L','T','X'] THEN
      CASE CH OF
        'E': EINGABE;
        'A': AUSGABE;
        'L': LOESCHEN;
        'T': TABELLE;
        'X': (* NICHTS *)
      END (* CASE *)
    ELSE
      WRITELN(CH ,' IST NICHT MÖGLICH!');
  UNTIL CH='X'
 END;
```

Listing 1. Kundenverwaltung mit Listenstruktur (Schluß). Das Zeichen »^« entspricht dem Hochpfeil (»↑«)

den Nachfolger des Records VOR↑ gesetzt. Dann kann der Zeiger NAECHSTER in VOR↑ auf den Wert des Zeigers NEU gesetzt werden. Damit ergibt sich eine Liste wie Bild 4c. Dort sind die geänderten Zeiger fett gezeichnet.

Um in der Prozedur AUSGABE zu einem Namen den zugehörigen Kundenrecord in der Liste zu finden, dient ebenfalls die Funktion VORHANDEN. Da sie jedoch immer einen Zeiger auf den Vorgänger liefert, muß der Record VOR↑.NAECHSTER verwendet werden.

Ebenso einfach ist das Löschen eines Kunden. In der Prozedur LOESCHEN wird wieder mit VORHANDEN ein Zeiger auf den Vorgänger in der Liste gesetzt. Mit

```
VOR↑.NAECHSTER:=VOR↑.NAECHSTER↑.
NAECHSTER
```

wird im Record VOR↑ der Zeiger auf den übernächsten Nachfolger gerichtet (fette Linie in Bild 4d).

Jetzt ist es an der Zeit, uns mit der Funktion VORHANDEN näher zu befassen. Sie durchsucht die alphabetisch sortierte Liste nach dem Namen S. Wird ein Record mit NAME=S gefunden, so ist VORHANDEN gleich TRUE. Der Zeiger Z weist dann auf den Vorgänger dieses Records. Ist jedoch VORHANDEN gleich FALSE, so existiert kein Kunde mit dem Namen S. Jetzt zeigt Z auf den Record, hinter dem ein neuer Record eingefügt werden müßte, um die alphabetische Ordnung zu erhalten. In Bild 4e ist gezeigt, daß zwei Zeiger Z und Z1 verwendet werden. Man durchläuft wie bei der Prozedur TABELLE die Liste bis gilt:

```
Z1↑.NAME >=S
```

Dabei hinkt der Zeiger Z immer einen Record hinter dem Zeiger Z1 her. Ist die obige Bedingung eingetreten, so zeigt Z auf den alphabetischen Vorgänger. Um zu verhindern, über das Listenende hinauszulaufen, wird am Anfang der letzte (unbenutzte) Record mit dem gesuchten Namen gefüllt:

```
ENDE↑.NAME:=S; (z.B. S='SCHULZE')
```

Damit bricht die WHILE-Schleife sicher für Z1=ENDE ab. Einen solchen Eintrag zur Vereinfachung des Abbruchkriteriums bezeichnet man als Marke. Das boolesche Ergebnis der Funktion bestimmt also abschließend die folgende Zuweisung:

```
VORHANDEN:= (Z1↑.NAME=S) AND
(Z1<> ENDE)
```

Am Programmanfang wird mit

```
NEW(KOPF); NEW(ENDE);
KOPF↑.NAECHSTER:=  ENDE
```

eine Liste mit zwei unbenutzten Records erzeugt. Durch diese Initialisierung findet Einfügen und Löschen immer zwischen zwei Records statt. Wären diese Hilfsrecords nicht vorhan-

den, so müßte man zum Beispiel beim Löschen immer die Sonderfälle Löschen am Listenanfang oder Löschen des einzigen Elements in der Liste prüfen, da in diesen Fällen auch die Zeiger KOPF und ENDE verändert werden müßten.

Jetzt sollten Sie sich zunächst etwas Zeit nehmen, alle diese Informationen zu verdauen und sich genauer mit dem Listing 1 auseinandersetzen. Dann führen Sie einen kleinen Verständnistest durch:

1. Was ist der Unterschied zwischen den folgenden Zuweisungen?

```
KOPF  := ENDE              und
KOPF↑ := ENDE↑?
```

2. Ändern Sie das Programm so, daß die Namen in umgekehrter alphabetischer Reihenfolge (SCHULZE, MUELLER, MAIER) in der Liste gespeichert werden!

Zur Beantwortung der ersten Frage blättern Sie bei Bedarf zum Anfang der Ausführungen über Zweigvariablen zurück. Die zweite Aufgabe besteht »nur« in der Änderung der Funktion VORHANDEN.

Es gibt sehr viele verschiedene Varianten mit dem Konzept der Speicherung von Daten in Listen. Sie unterscheiden sich in der Methode, wie die Records durch Zeiger verkettet sind. Im Beispiel der Kundenliste kann man ohne Probleme von jedem Kunden zu seinem alphabetischen Nachfolger gelangen. Jedoch erreicht man den Vorgänger im Alphabet nur, indem man die Liste vom Kopf her durchläuft. Eine naheliegende Lösung besteht darin, jeden Record um einen Zeiger auf den Vorgänger zu erweitern:

```
TYPE KUNDENZEIGER =  KUNDE;
     KUNDE = RECORD
       NAME : ARRAY
              [1..10] OF CHAR;
       KNUMMER  :  INTEGER;
       VORIGER  :  KUNDENZEIGER;
       NAECHSTER:  KUNDENZEIGER;
     END;
```

Diese Verkettung erlaubt zwar ein Durchlaufen der Liste vorwärts wie rückwärts, jedoch werden die Operationen zum Einfügen und Löschen wesentlich komplexer, da sie zwei Verkettungen aktualisieren müssen. Eine solche doppelt verkettete Liste zeigt symbolisch das Bild 5.

Es gibt noch viele andere Methoden, Datenstrukturen mit Zeigern zu bilden (zum Beispiel Bäume), in denen man sehr effizient Werte sortiert einfügen, suchen und löschen kann. Da dieses Themengebiet sehr umfangreich ist und wirklich gute Literatur über dieses Thema existiert [1], sollen die letzten Beispiele mit Zeigern andere Anwendungen zeigen.

Zunächst wird ein Verfahren vorgestellt, das Werte so speichert, daß man ohne Suchen in einem Schritt einen vorgegebenen Wert wiederfindet. Die angenommene Aufgabe besteht darin, für eine Anzahl von Orten die Postleitzahl abzulegen. Man soll also einerseits neue Orte mit ihrer Postleitzahl eingeben können und andererseits zu einem vorgegebenen Namen die gespeicherte Postleitzahl (falls gespeichert) erhalten.

Bei der Speicherung mit dynamischen Variablen im Programm KUNDENLISTE trat das Problem auf, daß man sich erst mit einer (linearen) Suche durch die Liste bewegen muß, um einen Kundenrecord zu finden. Dabei muß man bei n gespeicherten Werten im Durchschnitt n/2 Vergleiche aufwenden. Optimal wäre eine Speicherungsmethode, bei der man direkt aus dem Suchschlüssel einen Verweis auf die gespeicherten Informationen erhält. Diese Anforderung erfüllt das sogenannte »Hashing«: Man speichert alle Daten in einer Hash-Tabelle. Das ist ein Array, das mit ganzen Zahlen indiziert wird:

```
CONST HASHSIZE = 97;
TYPE INFO = INTEGER; (* POSTLEIT-
ZAHL *)
VAR HASHTAB  = ARRAY[0..HASH-
               SIZE] OF INFO;
```

Nun sollen im vorliegenden Beispiel als Schlüssel zwanzigstellige Städtenamen verwendet werden. Man steht also vor dem Problem, aus einem String der Länge 20 einen eindeutigen Index zwischen 0 und 1000 zu erzeugen. Eine solche Hash-Funktion - Schlüsseltransformations-Funktion - gibt das Programm in Listing 2 unter dem Namen HASHINDEX wieder. Zunächst wird die Summe aller Codezahlen der Zeichen berechnet:

```
F = 70
R = 82
A = 65
N = 78
K = 75
F = 70
U = 85
R = 82
T = 84
_ = 32
_ = 32
...
_ = 32
-------
  1043
```

Anschließend wird dieser Index durch die Tabellengröße geteilt. Der Divisionsrest ist eine Zahl zwischen 0 und der Tabellengröße und kann somit direkt als Index gelten.

```
1034 / 97 = 10 REST 73
```

Um die Postleitzahl für Frankfurt (6000) zu speichern, sind folgende Schritte erforderlich:

```
INDEX:=HASHINDEX('FRANKFURT    ');
HASHTAB[INDEX] := 6000
```

Index besitzt also den Wert 73. Genauso einfach ist es, die Postleitzahl für Frankfurt anzuzeigen:

```
INDEX:=HASHINDEX('FRANKFURT    ');
WRITELN('PLZ:', HASHTAB[INDEX];
```

Das Verfahren besitzt jedoch einen gravierenden Nachteil. Die Hashfunktion, die zu einem Städtenamen einen Index zwischen 0 und HASHSIZE bestimmt, muß nicht eindeutig sein. Es kann also durchaus sein, daß es einen weiteren Städtenamen (sagen wir X-Stadt) gibt, der bei der obigen Berechnung ebenfalls den Index 73 ergibt. Dann kommt es in der Hashtabelle zu einer Kollision (»hash clash«). In diesem Fall ist es mit der obigen Datenstruktur unmöglich festzustellen, ob in HASHTAB[73] die Postleitzahl von Frankfurt oder X-Stadt steht. Im Programm nach Listing 2 wird das bisherige Konzept deshalb folgendermaßen modifiziert:

```
TYPE HASHELEMENT = RECORD
                     NAME: STRING;
                     PLZ : INTEGER;
                     NEXT:↑HASHELEMENT
                   END;
VAR HASHTAB: ARRAY[0..HASHSIZE] OF
                     ↑HASHELEMENT;
```

Man speichert in der Hashtabelle nur einen Zeiger auf einen Record vom Typ Hashelement. Dieses Record enthält neben der eigentlichen Information (PLZ) noch den Name der Stadt. Ein Record vom Typ HASHELEMENT ist natürlich nur dann erforderlich, falls eine Postleitzahl gespeichert werden soll. Ansonsten besitzt der Zeiger in HASHTAB den Wert NIL. Diese Vorbelegung führt am Programmanfang die Prozedur LEERETABELLE durch. Das Feld NEXT im Record HASHELEMENT dient zur Behandlung von Kollisionen. Alle Schlüssel, die unter demselben Index stehen, werden in zufälliger Reihenfolge zu einer Liste mit dem Zeiger NEXT verkettet. Beim Einfügen und beim Abfragen muß deshalb eventuell diese Liste durchlaufen werden. Bild 6 zeigt die Datenstruktur der Hashtabelle. Jeder Zeiger im Array HASHTAB kann also Kopf einer Liste von Einträgen sein. Bitte nehmen Sie die Werte in der Abbildung nicht zu genau, da aus naheliegenden Gründen nicht für jeden Namen die Hashfunktion berechnet wurde.

Unter folgenden Bedingungen ist Hashing die mit Abstand beste Speicherungsform und jeder anderen Speichermethode in Geschwindigkeit und Programmieraufwand weit überlegen:

1. Es existiert eine Grenze für die Anzahl der Werte, die gespeichert werden sollen. Die Größe der Hashtabelle ist etwa um zehn Prozent größer als diese Maximalanzahl festzulegen.

2. Eine sortierte Ausgabe in der Reihenfolge der Schlüssel ist nicht erforderlich. Es werden nämlich die Schlüssel im Idealfall völlig ungeordnet über die Hash-Tabelle verteilt, so daß keine Möglichkeit existiert, von einem Eintrag zu seinem Nachfolger zu gelangen.

Aus dem bisher Gesagten ist klar, daß die Hash-Funktion alle Schlüssel (im Beispiel die Städtenamen) möglichst gleichmäßig über den gesamten Indexbereich verteilen soll. Dabei muß man natürlich vermeiden, daß häufig auftretende ähnliche Schlüssel denselben Index erhalten. Soll zum Beispiel die Häufigkeit von Variablennamen in Pascal bestimmt werden, so darf eine Hashfunktion keinesfalls alle Namen mit nur einem Buchstaben auf denselben Index abbilden, da sonst ständig eine lange Liste durchsucht werden muß.

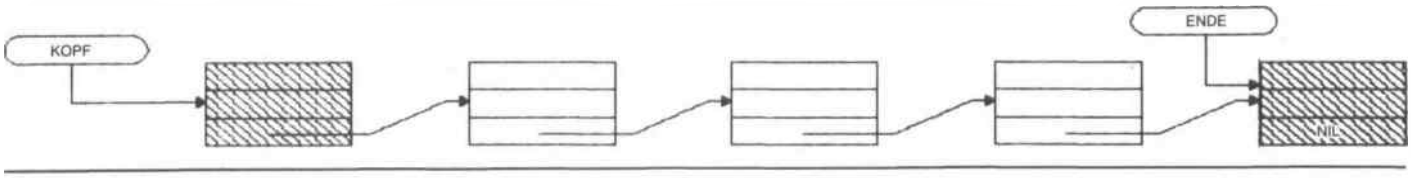


Bild 3. Kundenliste mit leerem Kopf und Ende

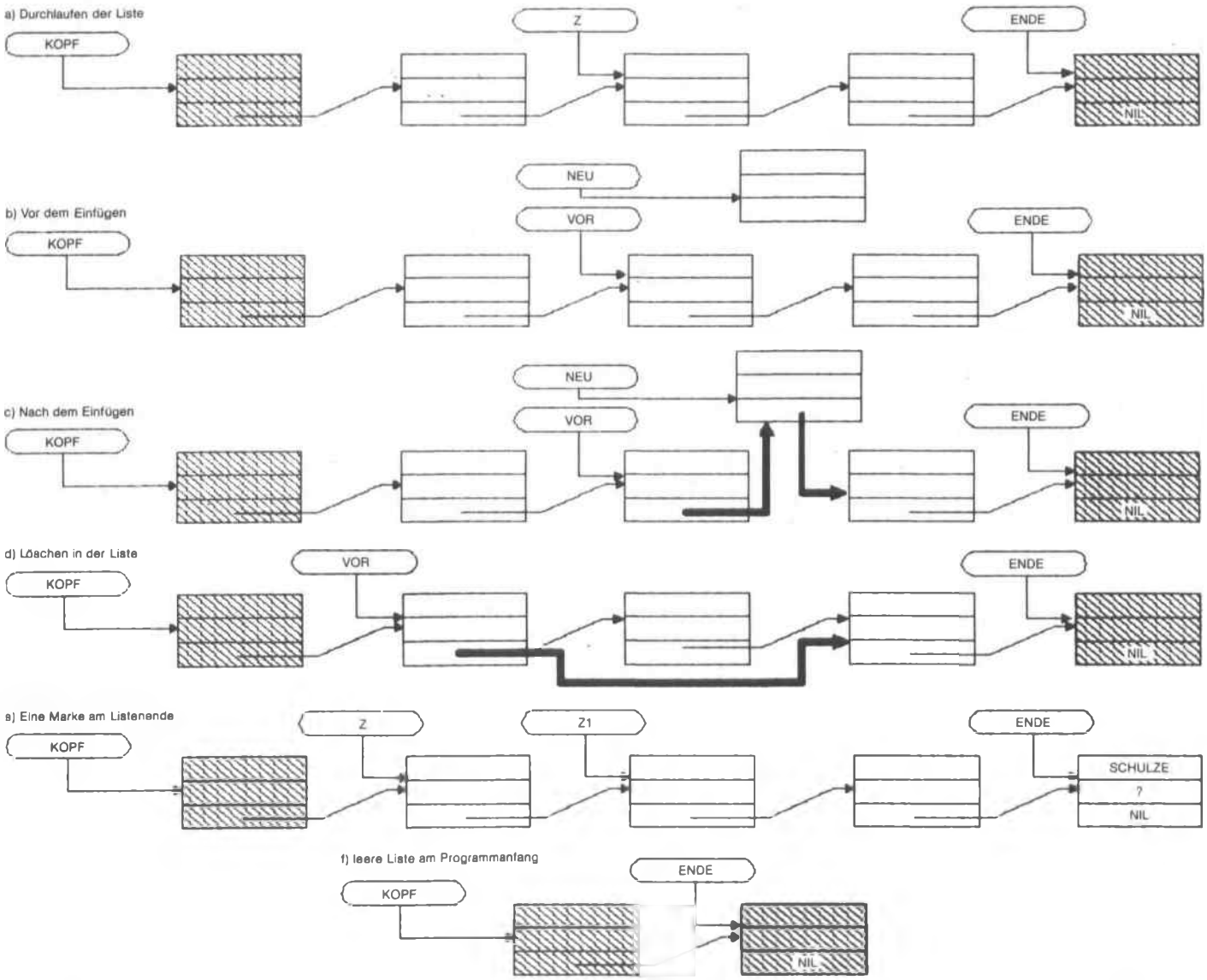


Bild 4. Die Kundenliste aus Listing 1

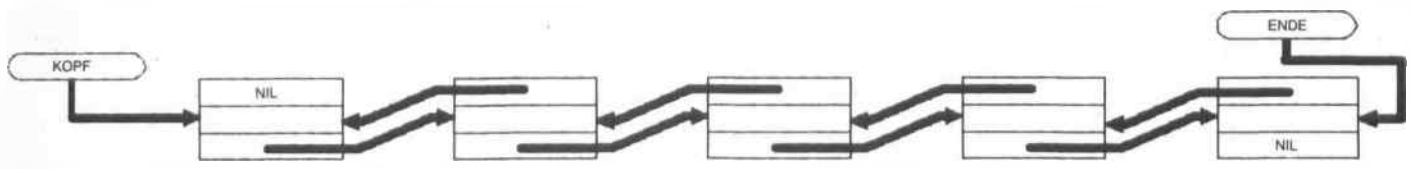


Bild 5. Eine doppelt verkettete Liste

Normalerweise ist die Hash-Funktion jedoch unkritisch, so daß man jede Information mit nur einem Zugriff auf einen Record erhält. Bemerkenswert ist noch die Speicherplatzökonomie des Verfahrens. Zwar ist ständig ein Array der Größe HASHSIZE vorhanden, dieses enthält jedoch nur Zeiger (zwei Byte Länge). Nur für tatsächlich vorhandene Schlüssel wird ein Record vom Typ HASHELEMENT dynamisch erzeugt, der neben der eigentlichen Information nur noch Speicherplatz für einen Zeiger (NEXT) benötigt. Unter den oben genannten zwei Bedingungen ist Hashing also wärmstens zu empfehlen, da es die Geschwindigkeit von Array-Zugriffen mit der Effizienz dynamischer Variablen verbindet.

Als abschließendes Beispiel steht an, eine Aufgabe mit Graphen in Pascal zu lösen. Ein Graph ist in der Mathematik ein abstraktes Gebilde aus Knoten und Kanten, die von Knoten zu Knoten führen. Uns interessiert eine spezielle Art von Graphen, nämlich sloche, deren Kanten gerichtet und markiert sind. Das klingt schrecklich abstrakt, läßt sich jedoch leicht mit einer Abbildung (Bild 7) erklären: Ein Knoten wird durch einen Kreis mit der Nummer des Knotens dargestellt, während ein Pfeil mit einem Zeichen zwischen zwei Knoten eine gerichtete, markierte Kante symbolisiert. Von Knoten 1 führt also eine Kante nach 2, die mit X markiert ist.

Nachdem nun die grundlegenden Begriffe bekannt sind, zur eigentlichen Aufgabe. Durch einen Graphen kann man einfache »Sprachen« beschreiben: Bild 8 zeigt einen Graphen, der alle Wörter der Form

AB
ABAB
ABABAB
ABABABAB..

»erkennt«. Dies geschieht folgendermaßen: Am Anfang setzt man eine »Marke« auf den Eingangsknoten 1. Außerdem gibt man das Wort vor, von dem man wissen will, ob es vom Graph erkannt wird:

ABAB

Nun liest man Buchstabe für Buchstabe und bewegt die Marke entsprechend den Zeichen an den Kanten durch den Graphen. Mit dem ersten Buchstaben des Wortes »ABAB« erreicht man von Knoten 1 den Knoten 2, da die Kante von 1 nach 2 mit dem Zeichen »A« markiert ist. Da der zweite Buchstabe ein »B« ist, wandert die Marke vom Knoten 2 entlang der Kante mit der Markierung »B« zum Knoten 3. Im dritten Schritt wird der dritte Buchstabe (wieder ein »A«) untersucht, so daß sich die Marke von Knoten 3 zurück zu Knoten 2 bewegt. Mit dem letzten Buchstaben »B« kommt die Marke

```
PROGRAM HASHING (INPUT, OUTPUT);
(* SPEICHERUNG VON POSTLEITZAHLEN MIT HASHING UND ÜBERLAUFLISTEN *)

  CONST HASHSIZE = 97; (* PRIMZAHL *)
        LEN = 20;

  TYPE  STRING = ARRAY [1..LEN] OF CHAR;
        HASHELEMENT = RECORD
                        NAME : STRING;
                        PLZ  : INTEGER;
                        NEXT : ^ HASHELEMENT
                      END;

  VAR   HASHTAB  : ARRAY[0..HASHSIZE] OF ^ HASHELEMENT;
        N        : STRING;
        PLZ      : INTEGER;

  PROCEDURE READSTRING (VAR S: STRING);
  (* STRING MIT LEN ZEICHEN VON DER TASTATUR LESEN *)
    VAR I: INTEGER;
        C: CHAR;
  BEGIN
    REPEAT READ(C) UNTIL C<>' '; (* VORLAUFENDE LEERZEICHEN IGNORIEREN  *)
    I:=1;

    REPEAT                          (* LEN ZEICHEN ODER BIS ZUM ZEILENENDE *)
      S[I]:= C; I:= I+1;
      READ(C)
    UNTIL (I>LEN) OR EOLN;

    WHILE I<LEN DO                  (* S MIT LEERZEICHEN AUF DIE VOLLE     *)
      BEGIN                         (* LÄNGE ERWEITERN                     *)
        S[I]:= ' '; I:= I+1;
      END;
    READLN
  END; (* READSTRING *)

  FUNCTION HASHINDEX (NAME: STRING): INTEGER;
  (* LIEFERT DEN INDEX IN DER HASHTABELLE FÜR DIESEN NAMEN *)
    VAR I     : 1..LEN;
        INDEX: INTEGER;
  BEGIN
    INDEX:= 0;
    FOR I:= 1 TO LEN DO INDEX:= INDEX + ORD(NAME[I]);
    HASHINDEX:= INDEX MOD HASHSIZE;
  END; (* HASHINDEX *)
PROCEDURE SPEICHERE (N: STRING; POSTLEITZAHL: INTEGER);
(* EINTRAG NAME MIT POSTLEITZAHL, ZUR SICHERHEIT TEST, OB DOPPELT *)

   VAR INDEX  : INTEGER;            (* INDEX IN HASHTAB            *)
       P,NEU  : ^HASHELEMENT;       (* ZEIGER IN ÜBERLAUFLISTE     *)
       DOPPELT: BOOLEAN;

BEGIN
  INDEX:= HASHINDEX(N);             (* INDEX DES NAMENS BERECHNEN  *)
  P:= HASHTAB[INDEX];               (* ZEIGER AUS DER HASHTABELLE  *)

  DOPPELT:= FALSE;                  (* PRÜFE AUF DOPPELTEN EINTRAG, *)
  WHILE P<>NIL DO                   (* FALLS AN DIESER POSITION    *)
    BEGIN                           (* BEREITS EINTRÄGE EXISTIEREN. *)
      DOPPELT:= DOPPELT OR (P^.NAME=N);
      P:= P^.NEXT
    END;

 IF DOPPELT THEN
     WRITELN(N, ' IST BEREITS GESPEICHERT!')
  ELSE
    BEGIN
      NEW(NEU);                          (* NEUEN RECORD ERZEUGEN      *)
      WITH NEU^ DO
        BEGIN
          NAME:=N; PLZ:= POSTLEITZAHL;(* WERTE EINTRAGEN             *)
          NEXT:= HASHTAB[INDEX]          (* VOR DEN ALTEN WERTEN EINFÜGEN *)
        END;
      HASHTAB[INDEX]:= NEU               (* NEU STEHT AN 1. POSITION   *)
    END; (* IF *)
END; (* SPEICHERE *)

FUNCTION POSTLEITZAHL(NAME: STRING): INTEGER;
(* LIEFERT DIE POSTLEITZAHL ZU DIESEM NAMEN ODER DIE ZAHL 0 *)
  VAR P     : ^ HASHELEMENT;
      INDEX: INTEGER;
BEGIN
  INDEX:= HASHINDEX(NAME);           (* BESTIMME DEN INDEX IN HASHTAB *)
  POSTLEITZAHL:= 0;                  (* VORLÄUFIG NOCH NICHT GEFUNDEN *)

  P:= HASHTAB[INDEX];                (* DURCHSUCHE DIE ÜBERLAUFLISTE  *)
  WHILE P<>NIL DO
    IF P^.NAME=NAME THEN
      BEGIN
        POSTLEITZAHL:= P^.PLZ;       (* FUNKTIONSERGEBNIS FESTLEGEN   *)
        P:= NIL                      (* ENDE DER SCHLEIFE ERZWINGEN   *)
      END
    ELSE
      P:= P^.NEXT;                   (* SONST WEITERSUCHEN            *)
 END; (* POSTLEITZAHL *)

 PROCEDURE LEERETABELLE;
```

Listing 2. Datenspeicherung mit Hashing

```
(* LÖSCHE DIE GESAMTE HASHTABELLE *)
  VAR I: 0..HASHSIZE;
BEGIN
  FOR I:= 0 TO HASHSIZE DO HASHTAB[I]:= NIL
END; (* LEERETABELLE *)
BEGIN (* HAUPTPROGRAMM *)
  LEERETABELLE;                          (* NOCH IST NICHTS GESPEICHERT     *)
  WRITELN('SPEICHERUNG VON POSTLEITZAHLEN: (BEENDEN MIT NAME = *)');

  WRITE('NAME: '); READSTRING(N);
  WHILE N[1]<>'*' DO
    BEGIN
      WRITE('PLZ :'); READ(PLZ);
      IF PLZ = 0 THEN                    (* SUCHE DIE ZUGEHÖRIGE PLZ:       *)
        WRITELN(POSTLEITZAHL(N))
      ELSE
        BEGIN
          WRITELN;
          SPEICHERE(N, PLZ);             (* SPEICHERE NAMEN MIT DIESER PLZ *)
        END;
      WRITE('NAME: '); READSTRING(N)
    END; (* WHILE *)
END.
```

Listing 2. Datenspeicherung mit Hashing (Schluß)

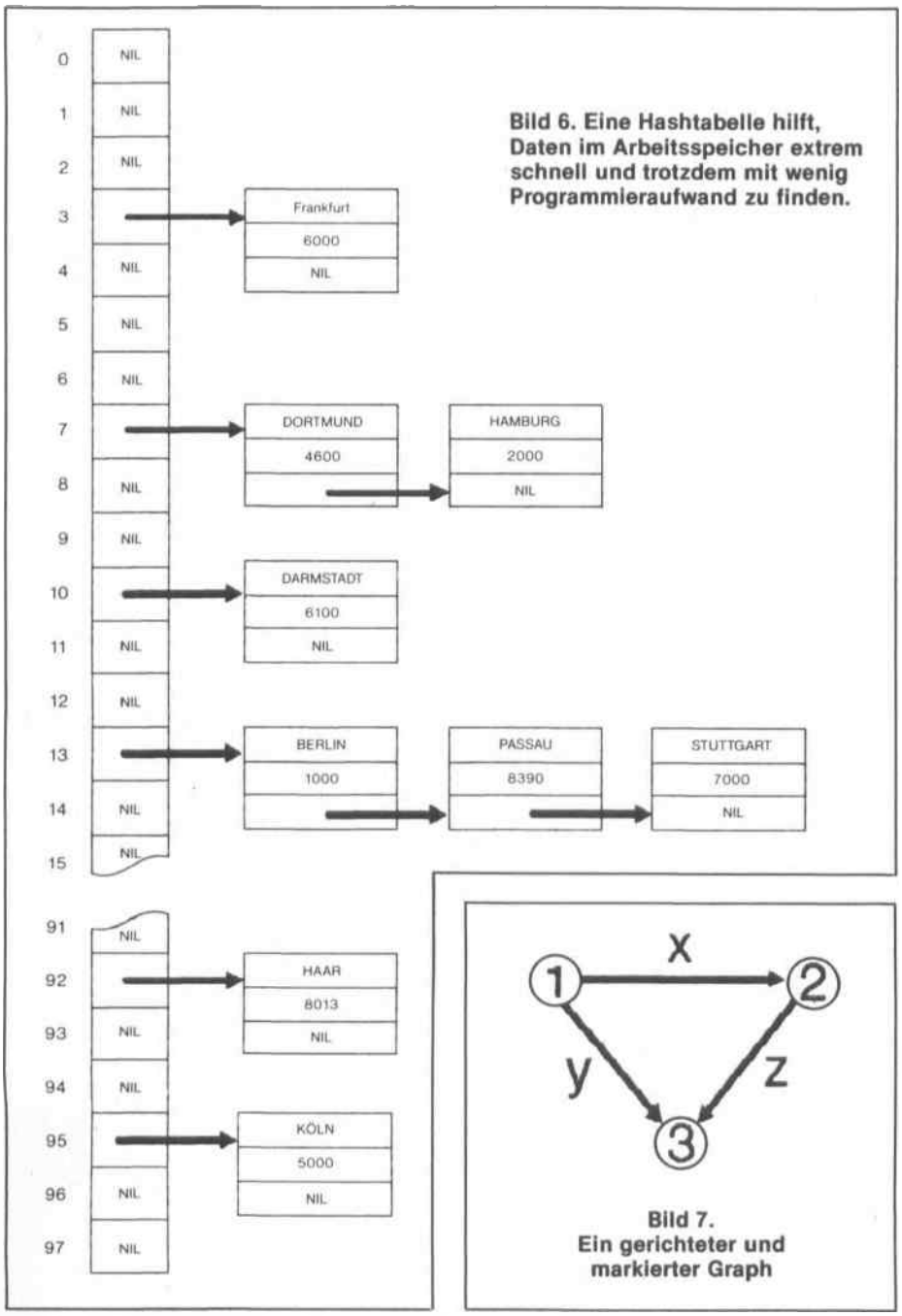


Bild 6. Eine Hashtabelle hilft, Daten im Arbeitsspeicher extrem schnell und trotzdem mit wenig Programmieraufwand zu finden.

Bild 7. Ein gerichteter und markierter Graph

schließlich auf Knoten 3 zu liegen. Da nun das Wort zu Ende ist, kann man aus der Position der Marke entscheiden, ob der Graph das Wort »erkannt« hat. Die Marke befindet sich im Knoten 3, den ein dicker Rand hervorhebt. Diese Knoten sind »akzeptierende« Knoten. Befindet sich die Marke am Schluß auf einem akzeptierenden Knoten, so ist das gesamte Wort erkannt und gehört somit zum Sprachschatz des Graphen. Sicher verstehen Sie jetzt auch, warum der Graph in Bild 8 alle Worte erkennt, die aus einer beliebig langen Folge von »AB« bestehen. Mit jedem »A« wird die Marke auf den Knoten 2 gesetzt, von wo aus mit »B« der akzeptierende Knoten 3 erreicht wird. Hätten wir jedoch das Wort

ABA

der obigen Testprozedur unterzogen, so wäre die Marke im dritten Schritt ebenfalls im Knoten 2 gelandet. Da das Wort an dieser Stelle bereits zu Ende ist, die Marke aber nicht auf einem akzeptierenden Knoten liegt, gehört »ABA« nicht zur Sprache des Graphen. Auch das Wort

ABAA

bleibt unerkannt, da mit ABA die Marke auf Knoten 2 liegt. Für den folgenden Buchstaben »A« existiert jedoch keine Markierung an einer Kante von 2 aus, so daß das gesamte Wort nicht erkannt wird.

Bild 9 zeigt einen Graphen, der alle Worte der Form

AAB
AABAAB
AABAABAAB
AABAABAABAAB...

identifiziert. Das können Sie bei der Anwendung der obigen Regeln für einige Beispielworte leicht herausfinden.

Listing 3 stellt ein Programm dar, das das mühsame Verfolgen des Weges der Marke durch den Graphen automatisch durchführt. Genauer gesagt soll das Programm folgendes leisten:

1. Es wird eine Beschreibung des Graphen eingelesen.
2. Es wird geprüft, ob überhaupt ein akzeptierender Knoten erreicht werden kann (siehe Bild 10 für ein Gegenbeispiel).
3. Für beliebig vorgegebene Wörter wird geprüft, ob diese erkannt werden.

Zunächst muß eine Datenstruktur für die interne Repräsentation des Graphen im Speicher des Computers gefunden werden. Da es beliebig viele Knoten und Kanten geben kann, wird man diese mit dynamischen Variablen darstellen. Graphen sind nämlich geradezu das klassische Beispiel für die Verwendung von Zeigern. Für jeden Knoten speichert man seine Nummer und eine Liste der Kanten, die von die-

sem Knoten wegführen. Jede Kante in dieser Liste enthält ihrerseits die Markierung und einen Zeiger auf den Knoten, zu dem sie führt.

```
TYPE TKNOTENREF =↑ TKNOTEN;
  TKANTENREF =↑ TKANTE;
  TKNOTEN     =  RECORD
        NUMMER: INTEGER;
        KANTENLISTE: TKANTENREF
       END;
  TKANTE      = RECORD
        MARKIERUNG : CHAR;
        NEXT: TKANTENREF;
        NACH: TKNOTENREF;
       END;
```

Es gibt also zwei Zeigertypen. Zeiger vom Typ TKNOTENREF zeigen immer auf Knoten (Records vom Typ TKNOTEN), während Zeiger vom Typ TKANTENREF immer auf Kanten (Records vom Typ TKANTE) weisen. Die Bilder 11 und 12 verdeutlichen jeweils die interne Darstellung eines Graphen. Die großen Kästen verbildlichen Records vom Typ TKNOTEN. Sie enthalten also die Nummer des Knotens. Diese Nummer ist negativ, falls es sich bei dem Knoten um einen akzeptierenden Knoten handelt. Außerdem ist jeder Knoten Kopf einer Liste von Records des Typs TKANTE (kleine Kästen). Für jede Kante wird das Zeichen, mit dem die Kante markiert ist, und ein Zeiger auf den Knoten am Ende der Kante, gespeichert. Da alle Kanten, die an einem Knoten beginnen, zu einer Liste verkettet sind, wird noch das Feld NEXT benötigt, in dem ein Zeiger auf die nächste Kante in der Liste enthalten ist. Von »außen« erreicht man den gesamten Graphen nur durch die Zeigervariable ANFANG.

Als erstes nun zur Funktion ERKANNT. Sie verwendet diese Datenstruktur, um zu prüfen, ob das Wort in dem Array W

```
VAR W: ARRAY[1..100] OF CHAR;
```

akzeptiert wird. Die Strategie ist sehr einfach und entspricht dem obigen Wandern einer Marke durch den Graphen. Der Parameter Q gibt die momentane Position der Marke (auf einem Knoten) an. I indiziert den momentan bearbeiteten Buchstaben im Wort W. Das Ende des Wortes markiert ein Dollar-Zeichen »$«. Wurde dieses Ende gelesen, so ist das Wort genau dann akzeptiert, wenn der momentane Knoten (Q↑) ein akzeptierender Knoten ist (Nummer ist negativ). Ansonsten wird die Kantenliste nach einer Kante mit der passenden Markierung durchsucht. Stimmt die Markierung mit dem laufenden Buchstaben W[I] überein, so setzt sich die Prüfung mit dem nächsten Buchstaben und dem Endknoten der Kante fort. Die Funktion ERKANNT ist also rekursiv. Bemerkenswert ist noch die Tatsache, daß von einem Knoten eventuell zwei oder mehrere gleich markierte Kanten

```
PROGRAM GRAPH (INPUT,OUTPUT);
(* DIESES PROGRAMM DEMONSTRIERT DIE VERWENDUNG VON DYNAMISCHEN VARIABLEN *)
(* ZUR DARSTELLUNG EINES GERICHTETEN, MARKIERTEN GRAPHEN IM SPEICHER DES *)
(* COMPUTERS. DIE PROZEDUR GRAPHEINLESEN BAUT EIN NETZWERK AUS RECORDS   *)
(* DER TYPEN TKNOTEN UND TKANTE AUF, DESSEN ERSTER KNOTEN ÜBER DEN       *)
(* ZEIGER ANFANG ERREICHT WERDEN KANN. DIE FUNKTIONEN ISTLEER UND        *)
(* ERKANNT BENUTZEN DIESE DATENSTRUKTUR, UM DIE DURCH DEN GRAPHEN BE-    *)
(* SCHRIEBENE SPRACHE NÄHER ZU UNTERSUCHEN (S. TEXT)                     *)

  CONST ENDE = '$';              (* MARKIERUNG AM ENDE DER EINGABE       *)

  TYPE  TEINGABE = CHAR;         (* DIE MARKIERUNGEN DER KANTEN BESTEHEN *)
                                 (* AUS EINZELNEN ZEICHEN                *)
        TKNOTENREF = ^TKNOTEN;   (* ZEIGER AUF EINEN KNOTEN              *)
        TKANTENREF = ^TKANTE;    (* ZEIGER AUF EINE KANTE                *)
        TKNOTEN = RECORD
                    NUMMER: INTEGER;    (* NEGATIV, FALLS AKZEPTIEREND   *)
                    ISTMARKIERT: BOOLEAN; (* FÜR FUNKTION ISTLEER        *)
                    KANTENLISTE: TKANTENREF; (* LISTE ALLER KANTEN, DIE  *)
                                       (* VON DIESEM KNOTEN WEGFÜHREN     *)
                    NEXT: TKNOTENREF;  (* VERKETTET ALLE KNOTEN IM GRAPH  *)
                                       (* ZU EINER LISTE                  *)
                   END;
         TKANTE = RECORD
                    MARKIERUNG: TEINGABE;
                    NACH: TKNOTENREF;  (* ZIELKNOTEN DIESER KANTE         *)
                    NEXT: TKANTENREF;  (* ZEIGER AUF DIE NÄCHSTE KANTE,   *)
                                       (* DIE AM SELBEN KNOTEN BEGINNT    *)
                   END;
  VAR  KNOTENLISTE: TKNOTENREF;    (* ANFANG DER LISTE ALLER KNOTEN       *)
       ANFANG     : TKNOTENREF;    (* ERSTER KNOTEN IM GRAPHEN            *)
       W: ARRAY [1..100] OF CHAR; (* WORT FÜR PRÜFUNG, OB ERKANNT        *)
       I: INTEGER;
  PROCEDURE GRAPHEINLESEN;
  (* KOMPLETTEN GRAPHEN IM SPEICHER AUFBAUEN. ANFANG (GLOBAL) ZEIGT AUF  *)
  (* DEN ERSTEN EINGEGEBENEN KNOTEN. FÜR JEDEN KNOTEN WIRD ISTMARKIERT:= *)
  (* FALSE GESETZT.  AKZEPTIERENDE KNOTEN WERDEN MIT NEGATIVEN NUMMERN   *)
  (* GESPEICHERT.                                                        *)
  (* DIE EINGABE VON DER TASTATUR HAT FOLGENDES FORMAT:  PRO ZEILE EINE  *)
  (* KANTE. NACHEINANDER NUMMER DES AUSGANGSKNOTENS, EIN ZEICHEN UNGLEICH*)
  (* LEERZEICHEN ALS MARKIERUNG UND DIE NUMMER DES ZIELKNOTENS.          *)
  (* DIE LETZTE ZEILE WIRD MIT EINER 0 BEENDET                           *)
     VAR ISTERSTER: BOOLEAN;     (* TRUE BEIM EINLESEN DER ERSTEN KANTE   *)
         VON      : INTEGER;     (* NUMMER DES AUSGANGSKNOTENS            *)
         NACH     : INTEGER;     (* NUMMER DES ZIELKNOTENS                *)
         MIT      : TEINGABE;    (* MARKIERUNG DER KANTE                  *)
         V,N      : TKNOTENREF;  (* ZEIGER AUF AUSGANGS- UND ZIELKNOTEN   *)
         KANTE    : TKANTENREF;  (* ZEIGER AUF NEU EINZUFÜGENDE KANTE     *)
     FUNCTION SUCHEKNOTEN (N: INTEGER): TKNOTENREF;
     (* SUCHE IN DER LISTE ALLER KNOTEN NACH EINEM KNOTEN, DER DIE NUMMER*)
     (* N BESITZT. DAS FUNKTIONSERGEBNIS IST EIN ZEIGER AUF EINEN RECORD *)
     (* VOM TYP TKNOTEN ODER DER WERT NIL, FALLS KEIN SOLCHER KNOTEN GE- *)
     (* FUNDEN WURDE.                                                    *)
        VAR Q: TKNOTENREF;
     BEGIN
       Q:= KNOTENLISTE;            (* ZEIGER AUF DEN ANFANG DER LISTE    *)
       SUCHEKNOTEN:= NIL;          (* FUNKTIONSERG. VORLÄUFIG FESTLEGEN  *)
       WHILE Q<>NIL DO             (* SOLANGE NOCH KNOTEN IN DER LISTE SIND *)
         IF Q^.NUMMER = N THEN     (* KNOTEN GEFUNDEN, ...               *)
           BEGIN
             SUCHEKNOTEN:= Q;      (* ZEIGER ALS FUNKTIONSERGEBNIS SETZEN *)
               Q:= NIL             (* SCHLEIFE BEENDEN                   *)
           END
         ELSE
           Q:= Q^.NEXT;            (* SONST WEITER MIT NÄCHSTEM KNOTEN IN *)
                                   (* DER LISTE                          *)
     END; (* SUCHEKNOTEN *)
     FUNCTION NEUERKNOTEN (N: INTEGER): TKNOTENREF;
     (* LEGE NEUEN (UNMARKIERTEN) KNOTEN MIT DER NUMMER N AN. DER RECORD *)
     (* WIRD IN DIE LISTE ALLER RECORDS DES GRAPHEN EINGEFÜGT, DAMIT ER  *)
     (* SPÄTER MIT DER FUNKTION SUCHEKNOTEN GEFUNDEN WIRD.               *)
        VAR Q: TKNOTENREF;
     BEGIN
       NEW(Q);                     (* SPEICHERPLATZ FÜR NEUEN RECORD HOLEN *)
       WITH Q^ DO                  (* RECORD KORREKT VORBELEGEN:         *)
         BEGIN
           NUMMER:= N;             (* NUMMER DES KNOTENS                 *)
           ISTMARKIERT:= FALSE;    (* NOCH UNMARKIERT                    *)
           KANTENLISTE:= NIL;      (* BIS JETZT BEGINNT NOCH KEINE KANTE AN *)
                                   (* DIESEM KNOTEN.                     *)
           NEXT:= KNOTENLISTE;     (* Q WIRD AM ANFANG DER KNOTENLISTE EIN- *)
         END;                      (* GEFÜGT.                            *)
       KNOTENLISTE:=Q;             (* Q IST JETZT ERSTER KNOTEN IN KNOTENL. *)
       NEUERKNOTEN:=Q;             (* FUNKTIONSERGEBNIS IST ZEIGER AUF Q^  *)
     END; (* NEUERKNOTEN *)
  BEGIN (* GRAPHEINLESEN *)
    KNOTENLISTE:= NIL;             (* NOCH IST DIE KNOTENLISTE LEER       *)
    ISTERSTER:= TRUE;              (* DIE FOLGENDE KANTE IST DIE ERSTE    *)
    READ(VON);
    WHILE VON<>0 DO                (* ENDE DER EINGABE WIRD DURCH KNOTEN MIT*)
      BEGIN                        (* DER NUMMER 0 GEKENNZEICHNET         *)
        REPEAT
          READ(MIT)                (* MARKIERUNG UNGLEICH LEERZEICHEN LESEN *)
        UNTIL MIT<>' ';
        READLN (NACH);             (* JETZT NOCH DIE NUMMER DES ENDKNOTENS *)
        V:= SUCHEKNOTEN(VON);      (* HOLE ZEIGER AUF DEN AUSGANGSKNOTEN  *)
        IF V=NIL THEN              (* AUSGANGSKNOTEN IST NEU, DESHALB     *)
          V:=NEUERKNOTEN(VON);     (* NEUEN KNOTEN ANLEGEN                *)
```

```
        N:= SUCHEKNOTEN(NACH);   (* HOLE ZEIGER AUF DEN ZIELKNOTEN          *)
        IF N=NIL THEN            (* FALLS NICHT BEREITS GESPEICHERT,        *)
          N:= NEUERKNOTEN(NACH);(* NEUEN KNOTEN ANLEGEN.                    *)
        NEW(KANTE);              (* BILDE JETZT EINE NEUE KANTE             *)
        KANTE^.MARKIERUNG:= MIT;(* EIN ZEICHEN ALS MARKIERUNG EINTRAGEN     *)
        KANTE^.NACH:= N;         (* KANTE ENTHÄLT ZEIGER AUF ZIELKNOTEN     *)
        (* JETZT KANTE IN DER LISTE DER KANTEN VON KNOTEN V^ EINFÜGEN:      *)
        KANTE^.NEXT:= V^.KANTENLISTE;
        V^.KANTENLISTE:= KANTE;
        IF ISTERSTER THEN        (* BEIM ERSTEN KNOTEN ANFANG SETZEN        *)
          BEGIN
            ISTERSTER:= FALSE;
            ANFANG:= V
          END;
        READ(VON)
      END; (* WHILE *)
    END; (* GRAPHEINLESEN *)
    FUNCTION ISTLEER (Q: TKNOTENREF): BOOLEAN;
    (* DIESE FUNKTION PRÜFT FÜR DEN KNOTEN Q^, OB KEIN AKZEPTIERENDER      *)
    (* KNOTEN ERREICHT WERDEN KANN. IN DIESEM FALL IST ISTLEER FÜR ALLE    *)
    (* VON Q ERREICHBAREN (NICHT MARKIERTEN) KNOTEN EBENFALLS TRUE         *)
    (* AM ANFANG MUSS DAS FELD ISTMARKIERT IM RECORD TKNOTEN FÜR ALLE      *)
    (* KNOTEN AUF FALSE GESETZT WERDEN.                                    *)
      VAR LEER : BOOLEAN;
          NACHF: TKANTENREF;
      BEGIN
        Q^.ISTMARKIERT:= TRUE;     (* MARKIERE KNOTEN, DAMIT DIESER NICHT   *)
                                   (* DOPPELT GEPRÜFT WIRD                  *)
        LEER:= Q^.NUMMER>=0;       (* FALLS Q SELBST EIN AKZEPTIERENDER     *)
                                   (* KNOTEN IST, KANN NATÜRLICH EIN AKZ.   *)
                                   (* KNOTEN ERREICHT WERDEN.               *)
        NACHF:= Q^.KANTENLISTE;    (* ZEIGER AUF DIE ERSTE KANTE, DIE BEI   *)
                                   (* KNOTEN Q BEGINNT                      *)
        (* SOLANGE KEIN AKZEPTIERENDER KNOTEN ERREICHT WURDE, VERFOLGE ALLE*)
        (* KANTEN, DIE VON Q WEGFÜHREN:                                     *)
        WHILE LEER AND (NACHF<>NIL) DO
          BEGIN
            IF NOT NACHF^.NACH^.ISTMARKIERT THEN
                                   (* FALLS ZIELKNOTEN DER KANTE UNMARKIERT *)
              LEER:= ISTLEER(NACHF^.NACH); (* PRÜFE DISEN ZIELKNOTEN      *)
            NACHF:= NACHF^.NEXT;   (* WEITER MIT DER NÄCHSTEN KANTE         *)
          END;
        ISTLEER:= LEER;            (* FALLS LEER IMMER NOCH TRUE IST, WIRD  *)
                                   (* ALSO KEIN AKZEPPT. KNOTEN ERREICHT    *)
    END; (* ISTLEER *)
    FUNCTION ERKANNT (Q: TKNOTENREF; I:INTEGER): BOOLEAN;
    (* PRÜFE, OB VON Q AUS MIT DER BUCHSTABENFOLGE AB W[I] EIN AKZEP-      *)
    (* TIERENDER KNOTEN ERREICHT WIRD.                                     *)
        VAR OK   : BOOLEAN;
            NACHF: TKANTENREF;
      BEGIN
        IF W[I] = ENDE THEN        (* ENDE DES WORTES W ERREICHT:           *)
          ERKANNT:= Q^.NUMMER<0    (* ERKANNT, FALLS Q^ EIN AKZEPTIERENDER  *)
                                   (* ZUSTAND IST.                          *)
        ELSE
          BEGIN
            OK:= FALSE;             (* NOCH IST KEIN AKZ. ZUSTAND GEFUNDEN *)
            NACHF:= Q^.KANTENLISTE;(* ZEIGER AUF DEN ANFANG DER LISTE DER   *)
                                   (* KANTEN, DIE BEI Q BEGINNEN            *)
            (* PROBIERE ALLE KANTEN, DIE BEI Q BEGINNEN UND MIT W[I]        *)
            (* MARKIERT SIND, BIS ERFOLG (OK=TRUE)                          *)
            WHILE NOT OK AND (NACHF<>NIL) DO
              BEGIN
                IF NACHF^.MARKIERUNG=W[I] THEN
                  OK:= ERKANNT(NACHF^.NACH, I+1);
                                   (* WEITER MIT DEM NÄCHSTE BUCHSTABEN     *)
                NACHF:= NACHF^.NEXT;(* NÄCHSTE KANTE IN DER LISTE            *)
              END;
            ERKANNT:= OK           (* FUNKTIONSERGEBNIS FESTLEGEN           *)
          END; (* IF *)
      END; (* ERKANNT *)
BEGIN (* HAUPTPROGRAMM *)
  GRAPHEINLESEN;                   (* HOLE BESCHREIBUNG DES GRAPHEN         *)
  WRITE('DIE AKZEPTIERTE SPRACHE IST ');
  IF NOT ISTLEER(ANFANG) THEN      (* PRÜFE, OB VOM ANFANGSKNOTEN EIN       *)
                                   (* AKZ. KNOTEN ERREICHBAR IST.           *)
    WRITE ('NICHT ');
  WRITELN ('LEER.');
  REPEAT
    WRITELN ('GEBEN SIE JETZT DAS ZU TESTENDE WORT EIN ($ AM WORTENDE)');
    WRITELN ('(PROGRAMMENDE MIT DEM WORT $)');
    I:= 0;
    REPEAT                         (* STRING W EINLESEN                     *)
      I:= I+1; READ (W[I]);
    UNTIL W[I]=ENDE;
    WRITELN; WRITE ('DAS WORT IST ');
    IF NOT ERKANNT(ANFANG,1) THEN  (* PRÜFE, OB MIT DEM WORT W VOM          *)
                                   (* ANFANGSKNOTEN EIN AKZ. KNOTEN         *)
                                   (* ERREICHBAR IST.                       *)
      WRITE('NICHT ');
    WRITELN('IN DER SPRACHE ENTHALTEN.');
  UNTIL W[1]='$';
END.
```

Listing 3. Das Programm zum Untersuchen von Graphen. Das Zeichen »^« entspricht dem Hochpfeil (»↑«).

ausgehen können. In diesem Fall prüft ERKANNT alle Wege, bis eine Kante zu einem Erfolg führt.

Ähnlich sieht die Lösung der Teilaufgabe 2 aus. Um festzustellen, ob überhaupt ein Wort erkannt wird, genügt es, eine beliebig markierte Kantenfolge vom Anfangsknoten zu einem akzeptierenden Knoten zu finden. Daher wird beim Aufruf ISTLEER für den Knoten Q, der als Parameter übergeben wird, zunächst geprüft, ob Q selbst ein akzeptierender Knoten ist. Wenn ja, so existiert ein akzeptiertes Wort. Ansonsten werden alle Kanten verfolgt, und geprüft, ob man über eine dieser Kanten einen Endzustand erreichen kann. Wiederum ruft sich als ISTLEER rekursiv auf. Jedoch würde ohne weitere Vorkehrung dieser Algorithmus beim Graphen aus Bild 10 in eine Endlosschleife geraten. Vom Ausgangsknoten 1 erreicht man den Knoten 2. Da man alle Kanten von 2 verfolgt, gelangt man über die Kante »B« zurück zu Knoten 1. Von dort springt man wieder zu Knoten 2 und so weiter. Die Lösung besteht darin, daß man alle bereits besuchten Knoten markiert. Dazu wird der Record um das Feld

```
ISTMARKERT: BOOLEAN
```

erweitert. Bei der Eingabe des Graphen sind alle Knoten unmarkiert. Erreicht man jedoch in der Funktion ISTLEER den Knoten Q, so wird dieser mit

```
Q↑ :ISTMARKERT:= TRUE
```

gekennzeichnet. Bevor man nun einen rekursiven Aufruf der Funktion ISTLEER ausführt, wird zunächst wieder geprüft, ob der Knoten nicht bereits untersucht und markiert wurde.

Das größte Problem ist sicherlich die Eingabe eines Graphen. In diesem Beispiel wird eine kantenorientierte Eingabe vorgenommen: Der Graph in Bild 8 wird folgendermaßen beschrieben:

```
 1 A  2
 2 B -3
-3 A  2
 0
```

Jede Zeile benennt also die Nummer des Anfangsknoten, die Markierung der Kante und die Nummer des Endknotens der Kante. Das Ende der Eingabe markiert die Zahl Null. Die Eingabe nimmt die Prozedur EINLESEN vor. Zunächst wird die Nummer des Ausgangsknotens (VON) gelesen, anschließend alle Leerzeichen ignoriert, bis die Markierung (MIT) indentifiziert wurde. Daran schließt sich die Nummer des Endknotens an. Akzeptierende Knoten sind wieder durch negative Zahlen kenntlich.

Für jeden Knoten wird zunächst festgestellt, ob bereits ein Record vom Typ TKNOTEN mit dieser Nummer existiert. Die Funktion SUCHEKNOTEN liefert zu einer Nummer N einen Zeiger auf einen

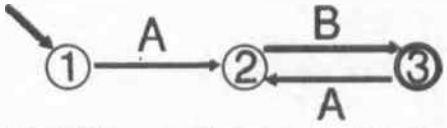


Bild 8. Dieser Graph kann eine einfache Zeichenfolge »erkennen«

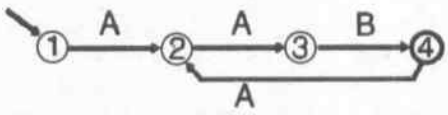


Bild 9. Ein Graph zum Erkennen einer weiteren Zeichenfolge

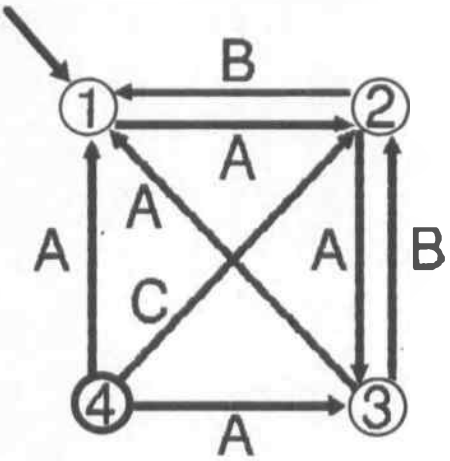


Bild 10. Dieser Graph akzeptiert überhaupt kein Wort

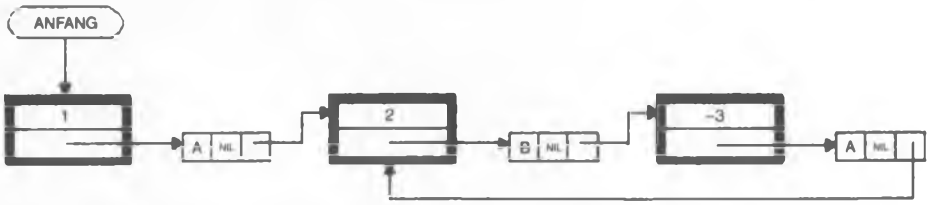


Bild 11. Interne Zeigerdarstellung des Graphen aus Bild 8

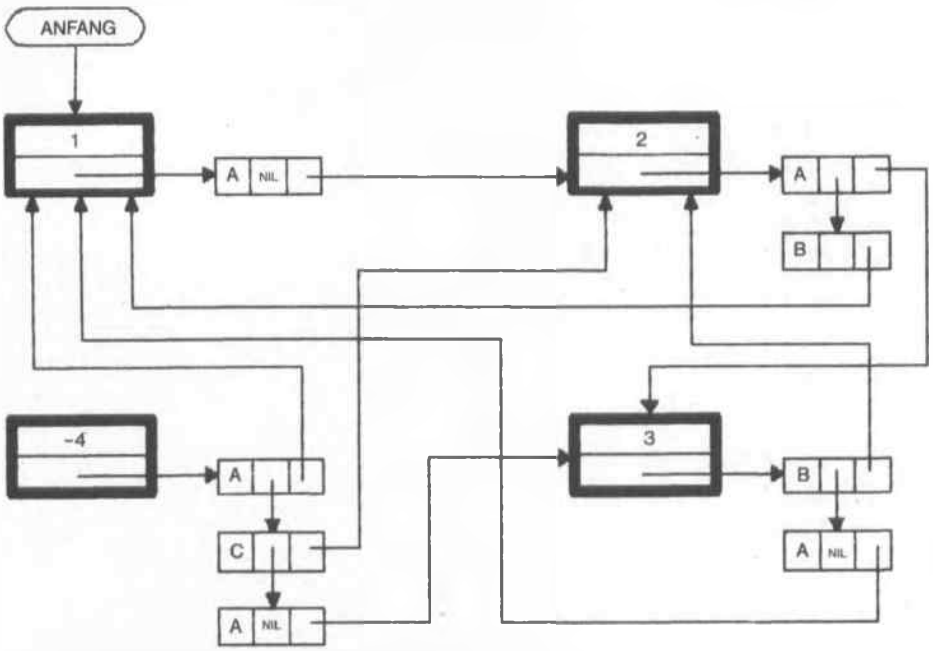


Bild 12. Interne Darstellung des Graphen aus Bild 10 im Speicher

```
 1 A 2
 2 B -3
-3 A 2
 0
DIE AKZEPTIERTE SPRACHE IST NICHT LEER.
ABAB$
DAS WORT IST IN DER SPRACHE ENTHALTEN
ABA$
DAS WORT IST NICHT IN DER SPRACHE ENTHALTEN
AA$
DAS WORT IST NICHT IN DER SPRACHE ENTHALTEN.
$
```

```
 1 A 2
 2 A 3
 3 B -4
-4 A 2
0
DIE AKZEPTIERTE SPRACHE SIT NICHT LEER
AAB$
DAS WORT IST IN DER SPRACHE ENTHALTEN.
AABAAB$
DAS WORT IST IN DER SPRACHE ENTHALTEN
A$
DAS WORT IST NICHT IN DER SPRACHE ENTHALTEN.
$
```

```
 1 A 2
 2 B 1
 2 A 3
 3 B 2
 3 A 1
-4 A 1
-4 C 2
-4 A 3
0
DIE AKZEPTIERTE SPRACHE IST LEER
$
```

Bild 13. Einige Probeläufe für den Graphen aus Bild 8

Knoten-Record, oder den Wert NIL, falls noch kein solcher Record vorkam. Eventuell muß also mit der Funktion NEUERKNOTEN ein Knoten mit der Nummer N angelegt werden. Die Funktion liefert einen Zeiger auf den neuen Record, der außerdem als unmarkiert gekennzeichnet wird.

Anschließend kann in EINLESEN ein neuer Record vom Typ TKANTE mit der angegebenen Markierung angelegt werden. Dabei wird im Feld NACH ein Zeiger auf den Endknoten N (mit der Nummer NACH) eingetragen und schließlich dieser Kanten-Record in die Liste der Kanten des Knotens V (mit der Nummer VON) eingefügt.

Da der Ausgangsknoten immer als erster in der Eingabe genannt wird, benutzt man die boolesche Variable ISTERSTER. Falls ISTERSTER=TRUE ist, muß also der Zeiger ANFANGSKNOTEN noch auf den Record V1 gesetzt werden.

Einige Probeläufe des Programmes mit verschiedenen Graphen zeigt Bild 13.

Natürlich ist nicht zu erwarten, daß ein absoluter Anfänger den gesamten Artikel sofort in die Praxis umsetzen kann. Die letzten Beispiele zeigen jedoch deutlich, daß Pascal für kompliziertere Aufgaben einen deutlichen Leistungsvorsprung gegenüber anderen Sprachen besitzt, die entweder nur einfache Datenstrukturen aufweisen (Basic, Fortran, Cobol), oder aber keine so wirkungsvolle Abstraktion von der internen Darstellung der Daten erlauben, so daß die Typüberprüfung in Ausdrücken und bei der Parameterübergabe auf den Programmierer abgewälzt wird (C und Forth).

(Florian Matthes/ev)

Info: Empfehlenswerte Pascal-Literatur:

F. Matthes, »Pascal mit dem C 64«, Markt&Technik Verlag (mit Pascal-Compiler aus Diskette)

N. Wirth, »Algorithmen und Datenstrukturen«, Teubner Studienbücher

K. Mehlhorn, »Effiziente Algorithmen«, Teubner Studienbücher

K. Jensen, N. Wirth, »Pascal, User Manual and Report, Lecture Notes in Computer Science, Vol. 18«, Springer Verlag (Berlin)

H. Schauer »Pascal für Anfänger«, R. Oldenbourg Verlag

E. Kaucher, R. Klatte, Ch. Ullrich, »Programmiersprachen im Griff, Band 2: Pascal 4, BI-Hochschultaschenbücher

R. Busch, »Der sichere Einstieg in Pascal«, Franzis' Verlag

R-D. Klein, »Was ist Pascal?«, Franzis' Verlag

J.N.P. Hume, R.C. Holt, »UCSD-Pascal«, Markt&Technik Verlag

D.T. Barnard, Crawford, »Pascal - Probleme und Anwendungen«, Markt&Technik Verlag

I. Lüke, P. Lüke, »Turbo Pascal«, Markt&Technik Verlag

»Turbo Pascal Handbuch«, Heimsoeth Software (mit Turbo Pascal Compiler auf Diskette)

»Turbo Tutor«, Heimsoeth Software (mit Demo-Diskette)

K.H. Rollke, »Das Turbo Pascal Buch«, Sybex-Verlag

A. Luehrmann, H. Pechham, »Appel II Pascal«, Tewi-Verlag

Prof. Nestle, E. Ostertag, »Kleiner Sprachführer Basic-Logo-Pascal«, Markt&Technik Verlag

K. Mikitta, »Pascal für Schulen«, Aulis-Verlag

I.R. Wilson, A.M. Addyman, »Pascal«, Carl-Hauser-Verlag

K.H. Rollke, »Grundkurs in Pascal, Band 1 und 2«, Sybex-Verlag

R. Zaks, »Einführung in Pascal und UCSD-Pascal«, Sybex-Verlag

J. Tiberghien, »Das Pascal Handbuch«, Sybex-Verlag

Cursor-Kur mit Turbo-Pascal

Der unstet flackernde Cursor des C 128 ist vielen CP/M-Anwendern ein Dorn im Auge. Hier naht Abhilfe in Form eines Turbo-Pascal-Programms.

Commodore hat beim neuen Basic 7.0 an den gestreßten Programmierer gedacht und gleich mehrere mögliche Cursor-Arten vorprogrammiert, die auf einfache Weise, nämlich durch Drücken der Escape-Taste und eines Buchstabens, für den Anwender erreichbar sind. So ist sowohl ein blinkender Strich- oder Unterstreichungs-Cursor wie auch ein stehender Block-Cursor machbar (siehe Tabelle).

Anders allerdings sieht es im CP/M-Modus des Computers aus. Hier sind dem Programmierer die Hände gebunden. Er muß sich wohl oder übel den närrisch flackernden Cursor gefallen lassen.

Unter CP/M sind Eingriffe in das Betriebssystem des Computers weitaus schwieriger zu realisieren als unter Basic. Für die Mehrheit der Commodore-Besitzer ist die Maschinensprache des Z80-Prozessors wohl mehr als ein Buch mit sieben Siegeln, so daß ein direkter Zugriff per Assembler ausscheidet. Außerdem erfordern Eingriffe in das Betriebssystem unter CP/M genaueste Kenntnisse des CP/M-Systems. Auch diese Kenntnisse dürften bei vielen Anwendern fehlen.

Aber dies ist trotzdem noch kein Grund zum Verzweifeln, denn wozu gibt es höhere Programmiersprachen? Aus obengenannter Not entstand das folgende Programm in Turbo-Pascal 3.0. Es bietet dem Benutzer die Möglichkeit, sich seinen Cursor nach eigenen Wünschen menügesteuert anzupassen. Das Programm kann nach dem Eingeben in ein COM-File compiliert werden, so daß man es ohne Probleme auf all seinen Disketten installieren kann. Es besteht sogar die Möglichkeit, das Programm von PROFILE.SUB aus aufzurufen, damit spart man sich das Laden nach dem BOOTen. Nach der Meldung »Cursor-Mode-Swap für C 128 und CP/M 3.0« kann man sich durch zwei oder drei Tastendrucke seinen persönlichen Cursor installieren.

Noch etwas zum Abtippen: Das Listing wurde im DIN-Modus der Tastatur eingegeben. Daher konnte zwar mit deutschen Umlauten gearbeitet werden, die geschweiften Kommentarklammern und die eckigen Mengenklammern wurden aber durch die von Turbo-Pascal ebenfalls zugelassenen Ersatzzeichen »(*« und »*)« (für geschweifte Klammern) beziehungsweise »(.« und ».)« (für eckige Klammern) ersetzt.

(Udo Reetz/ev)

Cursor-Mode	Escape-Sequenz
Block	ESC + S
Strich	ESC + U
Blinken	ESC + F
Stehen	ESC + E

Tabelle der möglichen Cursor-Formen im Basic 7.0

```
(*$u+*)

(*************************************************************)
(*   Beispiel für Zugriff auf die I/O-Ports des 8510:    *)
(* Utility zum komfortablen ändern des VDC Cursor Modes *)
(*      Turbo Pascal auf Commodore 128 unter CP/M        *)
(*************************************************************)

program cursorset (input, output);

var BlinkMaske,        (* Maske für Blinkbits in Reg. 10 *)
    StartLine: byte;   (* Rasterstartzeile des Cursors *)
            c: char;

procedure PortOut (adr: integer; wert: byte);
   (* Ausgabe von wert auf dem 16-Bit-Port adr *)
begin
  inline ($ed/$4b/adr/      (* LD   BC,(adr)  *)
          $3a/wert/         (* LD   A,(wert)  *)
          $ed/$79           (* OUT  (C),A     *)
         )
end (* procedure PortOut *);

procedure PortIn (adr: integer; var wert: byte);
   (* Lesen von Port adr nach wert *)
begin
  inline ($ed/$4b/adr/      (* LD   BC,(adr)  *)
          $ed/$78/          (* IN   A,(C)     *)
          $dd/$2a/wert/     (* LD   IX,(wert) *)
          $dd/$77/$00/      (* LD   (IX+0),A  *)
          $dd/$36/$01/$00   (* LD   (IX+1),0  *)
         )
end (* procedure PortIn *);

function vdcIn (reg: byte): byte;
   (* Liefert den Wert des VDC-Registers reg *)
var x: byte;    (* Hilfsvariable *)
begin
  PortOut ($d600, reg);     (* Registeradresse an VDC *)
  repeat
    PortIn ($d600, x)       (* Status lesen *)
  until x>=128;
  PortIn ($d601, x);        (* Daten-Register lesen *)
  vdcIn := x
end (* function vdcIn *);

procedure vdcOut (reg, wert: byte);
   (* Schreibt wert in VDC-Register reg *)
var x: byte;    (* Hilfsvariable *)
begin
 PortOut ($d600, reg);     (* Registeradresse an VDC *)
 repeat
    PortIn ($d600, x)       (* Status lesen *)
  until x>=128;
  PortOut ($d601, wert)     (* Daten-Register schreiben *)
end;

begin (* main *)
  clrscr;
  writeln ('Cursor-Mode-Swap für C 128 und CP/M 3.0': 20);
  writeln ('=================================': 20);
  writeln; writeln; writeln; writeln;
  write ('      Cursor B)linkend oder S)tehend ?');
  repeat
    read (kbd, c);
  until c in (. 'b', 'B', 's', 'S' .);
  writeln (c); writeln;
  if c in (. 's', 'S' .)
    then BlinkMaske := 0
    else
      begin
        write ('      S)chnell oder N)ormal blinken ? ');
        repeat
          read (kbd, c);
        until c in (. 's', 'S', 'n', 'N' .);
        writeln (c); writeln;
        if c in (. 's', 'S' .)
          then BlinkMaske := $40
          else BlinkMaske := $60
      end (* else *);
  write ('      U)nderline- oder B)lockcursor ? ');
  repeat
    read (kbd, c);
  until c in (. 'u', 'U', 'b', 'B' .);
  writeln (c); writeln;
  if c in (. 'u', 'U' .)
    then StartLine := 7
    else StartLine := 0;
  writeln; writeln;
  writeln ('**** Änderungen durchführen ? ****');
  repeat
    read (kbd, c)
  until c in (. 'j', 'J', 'n', 'N' .);
  writeln (c); writeln;
  if c in (. 'j', 'J' .)
    then
      begin
        vdcOut (10, BlinkMaske or StartLine);
        vdcOut (11,7)
      end (* else *)
end. (* program cursorset *)
```

Listing zu unserer Cursor-Kur

Kaiser

Möchten Sie auch einmal wie ein König ein Land regieren und das Staatsgeschick lenken? Dann versuchen Sie, als Kaiser Ihr Reich instand zu halten und für Wohlstand zu sorgen.

Kaiser oder auch Hammurabi ist eine Strategie-Simulation. Ihre Aufgabe ist es, durch Landkäufe oder -verkäufe, Verkauf von Getreide optimale Ausnutzung der Anbauflächen und gerechte Verteilung der Nahrungsmittel an die Bevölkerung das Reich in bestem Zustand zu erhalten. So kann zum Beispiel das Land zugrunde gehen, wenn die Bevölkerung nichts zu essen hat. Andererseits wollen die Leute eben mit vollem Bauch weniger arbeiten, so daß die Produktivität absinkt. Regeln Sie alles so, daß es Ihrem Reich wohler geht, so daß Sie lange regieren können.

Das Programm

Das Programm »Kaiser« (Listing) ist nur ein kleines, aber lauffähiges Grundgerüst, das ohne weiteres ausbaufähig ist. So können noch Faktoren wie arbeitsfaule Bevölkerung, Überfälle brandschatzender Horden, Unwetterkatastrophen oder vieles mehr mit integriert werden, um das Spiel reizvoller zu gestalten. Je komplexer die Faktoren sind, desto attraktiver wird auch das Spiel selbst.

Die Random-Funktion, mit der hier gearbeitet wird, ist nicht in jedem Pascal implementiert. Fehlt sie bei Ihnen, so müssen Sie noch eine Prozedur »random« mit in das Programm einbauen. Spielen Sie ruhig etwas mit dem Programm und bauen Sie von Zeit zu Zeit noch weitere Faktoren ein. Sie werden sehen: Durch viele Ereignisse, die das Geschehen beeinflussen können, gewinnt Kaiser immer mehr an Reiz.

(Dieter Mayer/hi)

```
PROGRAM kaiser;

CONST
   produktivitaet  = 7;

VAR
   anbauflaeche,gesamtflaeche,landkauf,
   vorrat,flaecheprokopf,rattenschaden,
   hektarertrag,ernte,zuteilung,nahrungprokopf,
   geburtenrate,sterberate,ausgaben,kornpreis,
   verkauf,vermoegen:real;
   jahr,jahre:0..20;
   zaehler,volk,zahl,geborene,
   gestorbene,bodenpreis:integer;
   getreide,kaufen,sinnvoll:boolean;
   input:char;

PROCEDURE initialisierung;
BEGIN
   zaehler        :=1;
   kornpreis      :=3;
   gesamtflaeche  :=30000;
   anbauflaeche   :=3000;
   bodenpreis     :=30;
   vermoegen      :=20000;
   hektarertrag   :=4;
   vorrat         :=20000;
   ernte          :=12000;
   volk           :=4000;
   flaecheprokopf:=gesamtflaeche/volk;
   geborene       :=0;
   gestorbene     :=0;
   nahrungprokopf:=20;
   ausgaben       :=vermoegen/8;
END;

PROCEDURE berechnung;
BEGIN
   IF kaufen = true THEN
    BEGIN
     gesamtflaeche:=gesamtflaeche+landkauf;
     vermoegen:=vermoegen-(landkauf*bodenpreis);
    END
                    ELSE
    BEGIN
     gesamtflaeche:=gesamtflaeche-landkauf;
     vermoegen:=vermoegen+(landkauf*bodenpreis);
    END;
   bodenpreis:=28+random(5);
   IF getreide = true THEN
    BEGIN
     vorrat:=vorrat-verkauf;
     vermoegen:=vermoegen+(kornpreis*verkauf)
    END;
   vorrat:=vorrat-(zuteilung*volk);
      ausgaben:=random(6000);
      vermoegen:=vermoegen-ausgaben;
      kornpreis:=1+random(3);
      hektarertrag:=4+random(3);
      ernte:=anbauflaeche*hektarertrag;
      vorrat:=vorrat+ernte;
      rattenschaden:=abs(random(50)/100);
      vorrat:=abs(vorrat*1-rattenschaden);
      flaecheprokopf:=gesamtflaeche/volk;
      geborene:=random(90);
      gestorbene:=random(25);
      volk:=volk+geborene-gestorbene;
END;

PROCEDURE zustandsausgabe;
   BEGIN
      writeln('Momentaner Spielstand:');
      writeln;
      write('Wir schreiben das ');
      IF zaehler>jahre THEN zaehler:=jahre;
      write(zaehler);
      writeln('. Jahr Ihrer Herrschaft.');
      writeln;
      write('Ihr Volk besteht zur Zeit aus ');
      write(volk);
      writeln(' Einwohnern, wobei Sie');
      write(geborene);
      write(' Geburten und ');
      write(gestorbene);
      writeln(' verstorbene Personen haben.');
      write('Sie besitzen ');
      write(gesamtflaeche:8:2);
      write(' Hektar Land, wovon ');
      write(anbauflaeche:8:2);
      writeln(' Hektar bebaut sind.');
      write('Jeder Buerger bewohnt durchschnittlich ');
      write(flaecheprokopf:6:2);
      writeln(' Hektar Boden.');
      write('Der Vorrat beläuft sich auf ');
      write(vorrat:10:2);
      writeln(' DZ Getreide.');
      write('Die Staatskasse enthaelt ');
      write(vermoegen:10:2);
      writeln(' Taler.');
      write('Staatsausgaben dieses Jahr: ');
      write(ausgaben:9:2);
      writeln(' Taler.');
      write('Heutiger Bodenpreis: ');
      write(bodenpreis);
      writeln(' Taler je Hektar.');
      write('Jahresertrag:  ');
      write(hektarertrag:8:2);
      writeln(' DZ Getreide je Hektar.');
      write('Kornpreis   :      ');
      write(kornpreis:5:2);
      writeln(' Taler je DZ.');
      write('Gesamternte :  ');
      write(ernte:8:2);
      writeln(' DZ Getreide.');
      write('Durch Ratten wurden ');
      write(rattenschaden:5:2);
      writeln(' % der Ernte vernichtet.');
     zaehler:=zaehler+1;
END;

PROCEDURE eingabe;
   BEGIN
      writeln;
      write('Moechten Sie Land kaufen ');
      write('oder verkaufen (K/V) ? ');
      read(input);
      writeln;
      IF input ='k' THEN kaufen:=true
                    ELSE kaufen:=false;
      write('Wieviel Hektar ? ');
      read(landkauf);
      writeln;
      write('Gedenken Sie, Getreide ');
      write('zu exportieren (J/N) ? ');
      read(input);
      writeln;
      IF input ='j' THEN
                      BEGIN
```

```
                        getreide:=true;
                        write('Wieviel DZ ? ');
                        read(verkauf);
                        writeln;
                      END
                    ELSE getreide:=false;
    write('Wieviele DZ Getreide goennen ');
    write('Sie jedem Untertan ? ');
    read(zuteilung);
    writeln;
    write('Wieviele Hektar duerfen ');
    write('die Bauern bewirtschaften ? ');
    read(anbauflaeche);
    writeln;
  END;

PROCEDURE eingabepruefung;
BEGIN
  writeln;
  IF kaufen = true THEN
    IF (landkauf*bodenpreis) >= vermoegen THEN
     BEGIN
       write('Landkauf erlaubt Ihre ');
       writeln('Staatskasse nicht.');
       sinnvoll:=false;
     END;
  IF kaufen = false THEN
  IF gesamtflaeche-landkauf <0 THEN
   BEGIN
     writeln('Soviel Land besitzen Sie nicht.');
     sinnvoll:=false;
   END;
 IF zuteilung <=0 THEN
   BEGIN
     write('Leuteschinder - Sollen ');
     writeln('die Buerger verhungern ?');
     sinnvoll:=false;
   END;
 IF zuteilung >= (vorrat/volk) THEN
   BEGIN
     writeln('So gross sind Ihre Vorraete nicht.');
     sinnvoll:=false;
   END;
  IF anbauflaeche <=0 THEN
    BEGIN
      write('Spassvogel - wo wollen ');
      writeln('Sie denn anbauen ?');
      sinnvoll:=false;
    END;
  IF anbauflaeche >0.5*(gesamtflaeche+landkauf) THEN
    BEGIN
      write('Nur die Haelfte des Landes ');
      writeln('kann bebaut werden.');
      sinnvoll:=false;
    END;
  IF anbauflaeche>produktivitaet*volk THEN
    BEGIN
      write('Soviel koennen Ihre Leute ');
      writeln('nicht bebauen.');
      sinnvoll:=false;
    END;
END;
BEGIN
  initialisierung;
  writeln('Wieviele Jahre moechten Sie regieren ?');
  readln(jahre);
  zahl:=random(99)+1;
  FOR jahr:=1 TO jahre DO
    BEGIN
      zustandsausgabe;
      REPEAT
        BEGIN
        sinnvoll:=true;
        eingabe;
        eingabepruefung;
        END;
      UNTIL sinnvoll;
      berechnung;
    END;
  zustandsausgabe;
END.
```

Listing »Kaiser«: Bestimmen Sie über das Wohl und Wehe Ihres Volkes

Berechnen gemeinsamer Teiler

Dieses Programm nimmt Ihnen die überaus lästige Arbeit ab, alle Zahlen herauszufinden, durch die eine andere teilbar ist.

Sicher hatten Sie schon gelegentlich mal das Problem, daß Sie alle Teiler einer Zahl herausfinden wollten. Sei es für die Verwendung im Bruchrechnen oder zur Ermittlung von Primzahlen. Diese Arbeit nimmt Ihnen nun die Prozedur »Teiler« (Listing) ab.

Nach der Compilation werden Sie gefragt, aus welcher Zahl alle Teiler berechnet werden sollen. Bitte geben Sie hier nur ganze Zahlen ein. Das Programm gibt Ihnen nun alle Teiler dieser Zahl aus. Wird kein Teiler gefunden, so ist es eine Primzahl, was auch ein Antwortsatz bekannt gibt.

Zum Programm

Nach der Eingabe der Zahl, aus der die Teiler berechnet werden sollen, wird zuerst ein Flag (Schalter) auf »unwahr« gesetzt. Danach versucht das Programm, die eingegebene Zahl so oft wie möglich zu dividieren. Sind Teiler vorhanden, werden diese ausgegeben und das Flag auf »wahr« gesetzt. Am Ende der Prozedur wird noch einmal der Zustand des Flags abgefragt. Ist es »unwahr«, so konnte kein Teiler gefunden werden, und die Meldung »Diese Zahl ist eine Primzahl« erscheint auf dem Bildschirm.

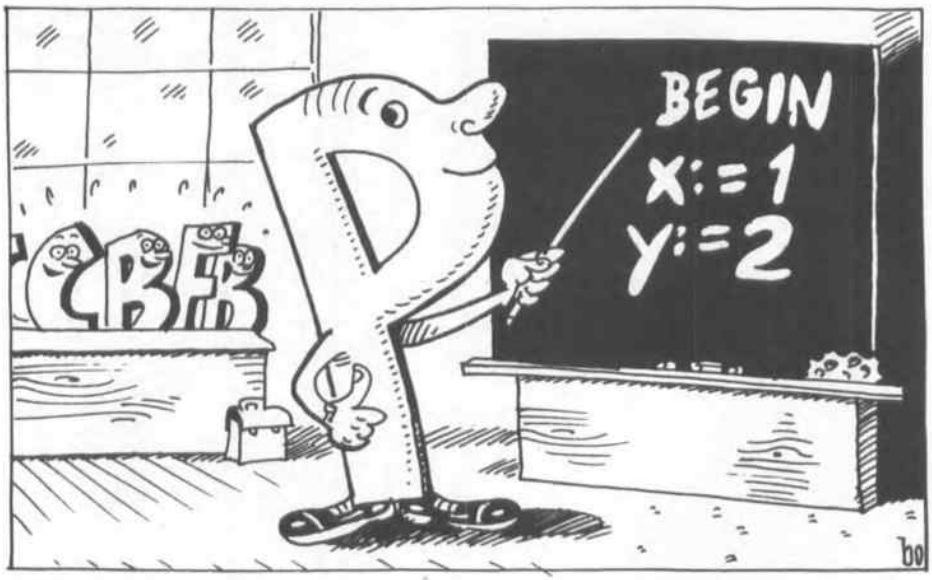
Die Prozedur eignet sich gut zum Einbinden in mathematische Programme, also etwa in ein Mathe-Lexikon auf dem Computer. (Dieter Mayer/hi)

```
PROGRAM Teiler (Input,Output);
VAR Zahl,Teiler:Integer;
    Schalter:boolean;
BEGIN
  write   ('Bitte geben Sie die Zahl ein,
             aus der die Teiler berechnet ');
  writeln ('werden sollen !');
  read (Zahl);
  Schalter:=FALSE;
WHILE Zahl>0 DO
    BEGIN
      FOR Teiler:=2 TO TRUNC(Zahl/2) DO
        BEGIN
          IF Zahl MOD(Teiler)=0
          THEN BEGIN write (Teiler:5);
                     Schalter:=TRUE
               END;
        END;
       IF Schalter=FALSE
        THEN write (' Diese Zahl
                     ist eine Primzahl');
       writeln; writeln;
       read (Zahl);
       Schalter:=FALSE;
    END;
END.
```

Listing »Teiler«

Filer für Turbo-Pascal

Werden lauffähige Turbo-Pascal-Programme erzeugt, so verbraucht die in jedem xxx.com-File vorhandene Run-Time-Library 8 KByte.



Sollen mehrere kurze Programme auf einer Diskette gespeichert werden, stößt man schnell an die Grenze der verfügbaren Speicher-Kapazität. Der Turbo-Pascal-Compiler bietet mit der H-Option die Möglichkeit, Programme ohne die Run-Time-Library zu compilieren und sie dann als sogenannte Chain-Files aufzurufen. Von dieser Möglichkeit, Platz zu sparen, macht das vorliegende Programm Gebrauch.

Die Benutzer-Programme werden speicherplatzsparend als xxx.chn-Files direkt auf die Filer-Diskette compiliert, ins Inhaltsverzeichnis (Inhalt.fil) eingelesen und dann von Filer aus aufgerufen. Um wieder zu Filer zurückkehren zu können, sollten die Benutzer-Programme mit der Prozedur »Ende« (siehe Listing) abgeschlossen werden.

Die Auswahl der einzelnen Funktionen von Filer erfolgt mit den Cursor-Tasten »Pfeil nach oben«, »..nach unten«, »..rechts und links«. Die jeweilige Funktion beziehungsweise die Benutzer-Programme werden invers als Leuchtfeld dargestellt. Durch Drücken der Home-Taste wird die angezeigte Funktion (Benutzer-Programm) ausgeführt.

»Programm« führt die Benutzer-Programme aus, »Neuaufnahme« liest ein auf der Diskette vorliegendes xxx.chn-Programm in das Inhaltsverzeichnis ein, »Löschen, Umbenennen und CP/M« erklären sich von allein und bedürfen somit keiner weiteren Erläuterung.

Ob groß, ob klein...

Alle vom Programm geforderten Eingaben können beliebig als Klein- oder Großbuchstaben erfolgen. Durch Drücken der ESC-Taste können alle laufenden Routinen abgebrochen werden.

Das Quellprogramm besteht aus filer.pas und lib.inc. In lib.inc können Anpassungen an verschiedene Terminals vorgenommen werden. Die Erläuterungen sind in das Programm-Listing eingefügt. Das lauffähige Programm besteht aus filer.com. Das File inhalt.fil wird beim ersten Programm-Lauf automatisch erzeugt.

(M.Schlösser/ev)

```
program filer;

(* Written by M.A. Schlösser       *)
(*            Rheinstr. 20         *)
(*            6100 Darmstadt       *)
(*            Tel: 06151 - 2 25 38 *)
(* Stand vom 25. Juni 1985         *)

(*****************************************************************)
(* Turbo-Pascal Programme können speicherplatz-sparend           *)
(* mit der H-Option des Compilers compiliert und von             *)
(* diesem Programm aus aufgerufen werden.                        *)
(*****************************************************************)

type inhaltrec = record
                 Progname: string (.12.);
                 end;

var   dateibuffer: inhaltrec;
      datei      : file of inhaltrec;
type
defString = array (.1..32.) of string (.12.);
defset    = set of char;
anystring = string (.80.);

const
bell      = ^G;  (* Glocke *)

var
lines, inharray        : defstring;
arraynr, arraylaenge,i : integer;
filename,helpstring    : anystring;
pruef,file_exist       : boolean;
filevar                : file;
zch                    : char;

(*$I filer.lib *)

procedure initinhalt (var z      : defstring;
                      var anz_rec: integer);
(* liest Inhaltsverz. von der Diskette und *)
(* erzeugt inharray                        *)
begin
anz_rec:= 0;
pruef:= exist ('a:inhalt.fil');
assign (datei,'a:inhalt.fil');
if pruef then
   begin
   reset(datei);
   while not eof (datei) do
         begin
         read (datei,dateibuffer);
         anz_rec:= anz_rec+1;
         z(.anz_rec.):= dateibuffer.progname;
         end; (* while *)
```

Listing. »filer.pas«

Fortsetzung auf Seite 68

```
     if anz_rec=0 then
        begin
        lowvideo;
     gotoxy(1,12);write ('Neuaufnahme aufrufen '); normvideo;
     end; (* if anz-rec *)
   end  (* if pruef *)
 else begin
     rewrite(datei);
     lowvideo;
     gotoxy(1,12);write ('Neuaufnahme aufrufen '); normvideo;
     end; (* else *)
close (datei);
end; (* procedure initinhalt *)


procedure head_string (var z:defstring);
(* Dient zur Zuordnung der Daten für den *)
(* auszudruckenden array                 *)

begin
z(.1.):= 'Programm'; z(.2.):= 'Neuaufnahme'; z(.3.):='Löschen';
z(.4.):= 'Umbenennen'; z(.5.):= 'CP/M';
end; (* procedure head_string *)


procedure programm_ueberschrift;
begin
gotoxy(1,16);clreol;
write ('PROGRAMME  (vorh: ',arraylaenge,'   max: 32 )');
end;


procedure msg(zeile:integer);
begin
gotoxy(1,zeile);
writeln ('Leuchtfeld mit Cursor-Tasten auf gewünschtes Programm stellen');
writeln ('und die HOME-Taste drücken.');
end;


procedure init_array;
(* Schreibt inharray auf Diskette *)

begin
assign (datei,'inhalt.fil');
rewrite (datei);
for i:= 1 to arraylaenge do
    begin
    dateibuffer.progname := inharray(.i.);
    write(datei,dateibuffer);
    end; (* for i *)
close (datei);
end;  (* procedure init_array *)

(*****************************************************)
(*       Hauptprogramm Filer                         *)
(*****************************************************)
begin
clrscr;
writeln('         Filer für Turbo-Pascal Programme unter CP/M');
writeln('         written by M.A.Schlösser          Juni 1985');
writeln;
for i:= 1 to 79 do write ('-');
initinhalt (inharray,arraylaenge);
head_string (lines); (* liest Funktion-Auswahl in das array lines *)
schreib_array (lines, 5,   1,5,     15,      4          );
          (* array/ anz/ x-y pos/ abstand/  worte/zeile *)
gotoxy(1,7); for i:= 1 to 79 do write ('-');
programm_ueberschrift;
schreib_array (inharray,arraylaenge,1,17,15,4);
(* schreibt Benutzer-Prog. *)
repeat (* until true = false *)
msg(9);
arraynr:= select (lines, 5,   1,5,     15,      4          );
             (* array/ anz/ x-y pos/ abstand/  worte/zeile *)
clrln(9,12);
case arraynr of
1: begin (* Programm *)
   gotoxy(1,9);lowvideo;
   writeln ('Programm-Auswahl');normvideo;
   msg(10);
   arraynr:= select (inharray,arraylaenge,1,17,15,4);
   if arraynr <> 0 then
      begin
      assign(filevar,inharray(.arraynr.)+'.chn');
      chain (filevar);
      end (* if arraynr *)
    else clrln(9,11);
   end; (* case of 1 *)
2: begin  (* Neuaufnahme *)
   gotoxy(1,9);Lowvideo;writeln('Neuaufnahme');normvideo;
   write ('Das ins  Inhaltsverzeichnis  aufzunehmende  File muß ');
   writeln ('unter xxx.chn auf der');
   writeln ('Diskette vorliegen. Die Namen-Eingabe ohne File-Kennung!');
   line_editor ('Programm-Namen eingeben',1,13,    8,     filename,
   ('0'..'9','A'..'z'.));
   (* Text-Ausgabe/ x-y pos/ Länge/ variable/ gültige Zeichen    *)
   if filename <> '' then
      begin
      Umw_in_Grossbuchstaben (filename);
      helpstring:= filename+'.chn';
      pruef:= exist (helpstring);
      if pruef then
         begin
         arraylaenge:=arraylaenge+1;
         inharray(.arraylaenge.):=filename;
         programm_ueberschrift;
         schreib_array (inharray,arraylaenge,1,17,15,4);
         init_array;
         end
      else begin
           write (bell);
           gotoxy(1,13);clreol;
           write ('Das File --> ',helpstring,'
                 <-- ist nicht auf Diskette');
          write (' vorhanden. Taste drücken.');
          repeat until keypressed;
          end; (* else *)
    end; (* if filename <> '' *)
    clrln (9,14);
    end; (* case of 2 *)
3: begin (* Löschen *)
   gotoxy(1,9);lowvideo; write ('Löschen'); normvideo;
   msg(10);
   arraynr:= select (inharray,arraylaenge,1,17,15,4);
   if arraynr<>0 then
          begin
          clrln(10,11);
          gotoxy(1,10);write ('File -->',inharray(.arraynr.),
                             '<-- löschen (j/n) ');
          repeat
          read (kbd,zch);
          until upcase(zch) IN ['J','N'];
          if upcase(zch) = 'J' then
             begin
             assign (filevar,inharray(.arraynr.)+'.chn');
             erase (filevar);
             for i:= arraynr to arraylaenge do
                 inharray(.i.):= inharray (.i+1.);
             arraylaenge:= arraylaenge-1;
             init_array;
             clrln(17,24);
             schreib_array (inharray,arraylaenge,1,17,15,4);
             end; (* if zch = J *)
          end; (* if arraynr <> 0 *)
       clrln(9,11);
       programm_ueberschrift;
       end; (* case löschen *)
    4: begin (* Umbenennen *)
```

Listing. »filer.pas«
(Fortsetzung)

```
   gotoxy(1,9);lowvideo; write ('Umbenennen'); normvideo;
   msg (10);
   arraynr:= select (inharray,arraylaenge,1,17,15,4);
   if arraynr <> 0 then
      begin
      file_exist:= false;
      clrln (10,11);
      gotoxy(1,10);
      writeln     ('File umbenennen :',inharray(.arraynr.));
      line_editor ('Neuer Name
                  ',1,11,8,filename,(.'0'..'9','A'..'z'.));
      Umw_in_Grossbuchstaben (filename);
      for i:= 1 to arraylaenge do
          if filename = inharray (.i.) then file_exist:= true;
      if file_exist then
         begin
         gotoxy(1,11);clreol;
         write (bell,'ein File ',filename,' existiert bereits.
               Taste drücken ');
         repeat until keypressed;
         end (* if file-exist *)
       else begin
            assign (filevar,inharray (.arraynr.)+'.chn');
            rename (filevar,filename+'.chn');
            inharray (.arraynr.):= filename;
            init_array;
            clrln (17,24);
            schreib_array (inharray,arraylaenge,1,17,15,4);
            end; (* else *)
      end; (* if arraynr <> 0 *)
   clrln (9,11);
   end; (* case umbenennen *)
5: begin (* Rückkehr zu CP/M *)
   clrscr; write ('CP/M - Betriebssystem');
   writeln;writeln;
   halt;
   end; (* case CP/M *)
end; (*case *)
until true=false;
end.
```

Listing. »filer.pas« (Schluß)

```
(* Library-include Proceduren *)
function exist(filename: anystring): boolean;
(* prüft, ob File auf Diskette vorhanden  *)
(* wenn vorhanden wird exist=true gesetzt *)
var f: file;
begin
   (*$I-*)
   assign(f,filename);
   reset(f);
   (*$I+*)
   if IOresult<>0 then exist:=false
   else exist:=true;
end; (* function exist *)
procedure Umw_in_Grossbuchstaben (var text:anystring);
(*$V-*)
var  i: integer;
begin
for i:= 1 to length(text) do
    text(.i.):= upcase (text(.i.));
end; (* procedure Umw_in_Grossbuchstaben *)
procedure schreib_array (var name:                    defstring;
                         anz,xstart,ystart,abstand,
                         worte_pro_Zeile:             integer);
(* schreibt array an vorgegebene Stelle des Bildschirms *)
(* defstring muss global definiert werden               *)
(* unter anz ist die Länge des Arrays zu verstehen      *)
var
i,x,y,wortzaehler : integer;
begin
x:= xstart; y:=ystart; wortzaehler:= 0;
gotoxy(xstart,ystart);clreol;
for i:= 1 to anz do
    begin
    gotoxy(x,y);
    write (name(.i.));
    x:= x+abstand;
    wortzaehler:=wortzaehler + 1;
    if wortzaehler= worte_pro_zeile
    then begin
         y:= y+1; x:= xstart; wortzaehler:= 0;
         gotoxy(x,y); clreol;
         end;
    end; (* for i *)
end; (* procedure schreib_array *)

function select   (var name                          : defstring;
                       anz,xstart,ystart,abstand,
                       worte_pro_zeile               : integer): integer;
(* Bewegt das Leuchtfeld entsprechend der definierten Tasten     *)
(* wird die ESC-Taste gedrückt, so wird select=0 zurückgeliefert *)
(* ansonsten die lfdnr des übergebenen Arrays                    *)
const
(* Diese Codes beziehen sich auf das Terminal *)
(* Basis 100 und müssen auf andere Terminals  *)
(* angepasst werden.                          *)

hoch =  #139;  (* Leuchtfeld 1 Zeile nach oben  *)
runter= #138;  (* Leuchtfeld 1 Zeile nach unten *)
rechts= #149;  (* Leuchtfeld 1 Wort nach rechts *)
links=  #136;  (* Leuchtfeld 1 Wort nach links  *)
bell  = ^G;    (* Glocke                        *)
esc   = #27;   (* ESC - Taste                   *)
home  = #192;  (* HOME - Taste                  *)
var
zch              : char;
x,y,xmax,ymax,i,
arraynr          : integer;
on_off           : boolean;

procedure invers_on_off;
begin
gotoxy(x,y);
for i:= 1 to abstand do
   write (' ');
on_off:= not (on_off);
if on_off then lowvideo;
gotoxy(x,y);
write (name(.arraynr.));
normvideo;
end; (* procedure invers_on_off *)
(* Hauptprogramm selector *)
begin
x:= xstart; y:= ystart; arraynr:= 1; on_off:= false;
xmax:= worte_pro_zeile * abstand;
ymax:=( anz div worte_pro_zeile)+ystart;
if anz mod worte_pro_zeile > 0 then ymax:=ymax + 1;
invers_on_off;
repeat
   repeat
   read (kbd,zch);
   until zch IN (.hoch,runter,rechts,links,esc,home.);
case zch of
hoch: begin
      invers_on_off;
      y:=y-1;
      if y<ystart then
         begin
         y:= ystart;
         invers_on_off;
         end (* if *)
      else begin
           arraynr:= arraynr - worte_pro_zeile;
           invers_on_off;
           end;
      end; (* case hoch *)
```

Listing. »lib.inc«

```
runter: begin
        invers_on_off;
        y:=y+1;
        if y>ymax then
           begin
           y:= y-1;
           invers_on_off;
           end (* if *)
        else begin
             arraynr:= arraynr + worte_pro_zeile;
             if arraynr> anz then
                begin
                arraynr:= arraynr-worte_pro_zeile;
                y:=y-1;
                end;
             invers_on_off;
              end; (* else *)
         end; (* runter *)
rechts: begin
        invers_on_off;
        x:=x+abstand;
        if x>xmax then
           begin
           x:= x-abstand;
           invers_on_off;
           end
        else begin
             arraynr:= arraynr + 1;
             if arraynr > anz then
                begin
                arraynr:= anz;
                x:= x-abstand;
                end;
              invers_on_off;
             end; (* else *)
        end; (* rechts *)
links : begin
        invers_on_off;
        x:=x-abstand;
        if x<xstart then
           begin
           x:= xstart;
           invers_on_off;
           end
        else begin
             arraynr:= arraynr - 1;
             if arraynr<1 then
                begin
                 arraynr:= 1;
                 x:= xstart;
                 end;
              invers_on_off;
             end; (* else *)
        end; (* links *)
esc : begin
      invers_on_off;
      arraynr:= 0;
      zch:= home;
      end; (* esc *)
end; (* case *)
until zch=home;
if arraynr<> 0 then invers_on_off;
select:= arraynr;
end; (* function select *)
procedure line_editor (text          : anystring;
                       xstart,ystart,
                       eingfeld      : integer;
                       var stringvar : anystring;
                       addstring     : defset);
(* Wird die ESC-Taste gedrückt, so wird ein leerer String    *)
(* zurückgeliefert. Unter eingfeld ist die Länge des zurück- *)
(* zuliefernden Strings zu verstehen.                        *)
(*$V-*)
const   (* Diese Codes überprüfen und evtl. anpassen    *)
        bell    = ^G;    (* Klingelzeichen              *)
        Esc     = #27;   (* ESC-Taste                   *)
        return  = #13;   (* Return Taste                *)
        backspc = ^H;    (* Cursor 1 Zeichen nach links *)

var     x,i,
        zaehler  : integer;
        zch      : char;

begin
zaehler:= 0; stringvar:=''; x:= length (text) +3;

(* Schreibe Text und stelle Cursor auf  *)
(* Eingabefeld                          *)
gotoxy (xstart,ystart);clreol; write (text,' :');
for i:= 1 to eingfeld do
       write (' ');
write (':');
gotoxy(x,ystart);

(* Erzeugen der auszugebenden Variablen *)
repeat (* until return *)
  repeat
  read (kbd,zch);
  until (zch IN (.backspc,return,esc.)) or (zch IN (addstring));
if (zaehler >= eingfeld) and (not(zch IN (.backspc,return,esc.))) then
   write (bell)
else begin
     case zch of
     backspc: begin
              if length(stringvar)>0 then
                 begin
                 delete (stringvar,length(stringvar),1);
                 write (backspc,' ',backspc);
                 zaehler:= zaehler-1;
                 end;
               end; (* backspc *)
     Esc: begin
          stringvar:= '';
          zch:= return;
          end; (* esc *)
     return::
     else
     stringvar:= stringvar+zch;
     write (zch);
     zaehler:= zaehler + 1;
     end; (* case *)
     end; (* else *)
     until zch = return;
     end; (* procedure line_editor *)
     procedure clrln (start,ende:integer);
     (* löscht die Zeilen von Start bis Ende *)
     var  i: integer;
     begin
     for i:= start to Ende do
         begin
         gotoxy(1,i);clreol;
         end;
     end; (* procedure clrln *)
     (* Ende der Library-include Proceduren *)
```

Listing. »lib.inc« (Schluß)

```
     (* Die Benutzer-Programme xxx.chn sollten    *)
     (* mit dieser Procedur abgeschlossen werden, *)
     (* um nach Filer zurückkehren zu können      *)

     procedure ende;
     var   diskfile : file;
     begin
     assign (diskfile,'filer.com');
     execute (diskfile);
     end;
```

Listing. »ende.inc«

Zeichenfolge analysiert

Nicht nur zum Compilerbau müssen Zeichenketten untersucht werden. Wie man so etwas macht, das lesen Sie hier.

Formale Sprachen, wie zum Beispiel alle Programmiersprachen, sind künstliche Sprachen. Sie werden jedoch nach den gleichen Gesetzen wie unsere natürlichen Sprachen gebildet. Dazu muß man sich über einige Grundbegriffe Klarheit verschaffen.

Vokabular = Menge der existierenden Wörter
Satz = Folge von Wörtern
Syntax (Grammatik) = Regel zur Bildung von korrekten Sätzen

Die Syntax legt fest, welche Wortfolgen zu einer Sprache gehören. Genau dies muß auch beim Compilerbau beachtet werden, da der Compiler beim Übersetzen überprüfen muß, ob ein Programm syntaktisch richtig geschrieben ist.

Die beiden nachfolgenden Programme lösen diese Aufgabe für eine sehr einfache formale Sprache.
Gegeben ist folgende Grammatik:
<A> : : = a (<A>)
Damit gelten zum Beispiel folgende Sätze:
<A> → ((a))
<A> → a
<A> → (((a)))

Das bedeutet, daß unsere Sprache nur drei Zeichenelemente kennt »a«, »(« und »)«. Das Bildungsgesetz wird aus den Beispielen oben deutlich: entweder handelt es sich um das Zeichen »a«, oder, falls Klammern gesetzt sind, muß die Anzahl der linken Klammern ebenso groß sein wie die der rechten.

Lösung 1: rekursive Analyse

Die Aufgabe wird in dem ersten Listing mit Hilfe einer rekursiven Prozedur gelöst. In der Prozedur wird jede öffnende Klammer auf dem Stack abgelegt. Ist das »a« erreicht, sozusagen der »Wendepunkt«, wird durch die schließenden Klammern der Stack wieder geleert.

Falls an irgendeiner Stelle ein Fehler festgestellt wurde, so setzt der Computer ein »Fehlerflag« (FEHLER : BOOLEAN).

Die Grundstruktur zur Feststellung des übergebenen Zeichens stellt ein CASE-Konstrukt dar, in dem vier Fälle unterschieden werden:

1. Zeichen = »(«
2. Zeichen = »a«
3. Zeichen = »)«
4. alle anderen Zeichen (OTHERWISE)

Im ersten Fall müssen zwei grundsätzliche Aufgaben gelöst werden: a) Rekursiver Prozeduraufruf mit einem eben gelesenen Zeichen und b) nach der Rückkehr von der höheren Rekursionsebene die Überprüfung, ob zu der öffnenden Klammer der Prozedurebene das nächste Zeichen, welches eine schließende Klammer sein sollte, paßt. Im Fehlerfall wird an den Variablenparameter FEHLER (Fehlerflag) die »Falsch«-Information weitergegeben. Im zweiten Fall bleibt, da »a« ein erlaubtes Zeichen ist, aber danach nur mehr das Leeren des Stacks stattfindet, der Anweisungsteil leer. Der dritte Fall entspricht wiederum dem ersten und der vierte und letzte Fall ist sehr einfach, da hier nur das Fehlerflag gesetzt werden muß (unzulässiges Zeichen).

Im Hauptprogramm werden nach dem Bildschirmaufbau und der Eingabeaufforderung zunächst durch den Konstrukt

```
REPEAT
READ(ZEICHEN);
UNTIL ZEICHEN <> ' ';
```

führende Leerzeichen überlesen, wodurch nach dem Verlassen der Schleife in ZEICHEN schon das erste Zeichen steht. Als nächstes wird das Fehlerflag mit FALSE initialisiert. Anschließend kann nun die Untersuchung des ersten eingelesenen Zeichens in einem CASE-Konstrukt folgen.

Für den Fall einer öffnenden Klammer wird sofort mit diesem Zeichen und dem Fehlerflag die Prozedur aufgerufen, die sich wiederum selbst so oft aufruft, bis das aktuelle Zeichen ein »a« ist. Danach werden die auf den Stack abgelegten öffnenden Klammern wieder abgetragen. Ist das Zeichen ein »a«, passiert nichts, denn das Ganze ist schon fast fertig.

```
program zeichen_rekursiv (input,output);

(*****************************************)
(***                                   ***)
(*** Rekursive Analyse einer Grammatik ***)
(***                                   ***)
(*****************************************)

var zeichen : char;
    fehler  : boolean;

procedure a (z : char; var fehler : boolean);

(*************************************)
(***                               ***)
(*** Rekursive Analyse Precedure ***)
(***                               ***)
(*************************************)

begin
   case z of
      '(' : begin
               read(z);
               a(z,fehler);
               read(z);
               if (z <> ')') then fehler := true;
            end;
    'a' : ;
    otherwise
       fehler := true;
   end; (* case z of *)
end; (* Procedure a *)

begin

(**********************)
(***                ***)
(*** Hauptprogramm ***)
(***                ***)
(**********************)

   writeln;
   writeln (' Eingabe ');
   readln;
   repeat
      read(zeichen);
   until (zeichen <> ' ');
   fehler := false;
   case zeichen of
      '(' : a (zeichen,fehler);
      'a' : ;
      otherwise
         fehler := true;
   end; (* case zeichen of *)
   read(zeichen);
   if (zeichen <> ' ') then
      fehler := true;
   writeln;
   if fehler
      then
         writeln ('* Fehler *')
      else
         writeln ('# O.K. #');
   writeln;
end.
```

Listing 1. Rekursion ist die elegante Lösung

Lediglich wenn noch ein anderes Zeichen gelesen wurde, wird ein Fehlerfall durch das Fehlerflag angezeigt. Um auszuschließen, daß durch ein noch folgendes Zeichen das Eingabewort noch unerkannterweise falsch wäre, wird dies eingelesen und überprüft: ist es ungleich einem Leerzeichen, wird wiederum das Fehlerflag aktiv. Den Schluß bildet noch ein IF-THEN-ELSE-Konstrukt, das dem Benutzer aufgrund der Variablen FEHLER mitteilt, ob das Eingabewort syntaktisch korrekt ist oder nicht.

Lösung 2 : Iterative Analyse

Dieses Programm löst die Syntaxüberprüfung auf »konventionellem« Wege. Dazu werden zunächst wie bei der rekursiven Lösung erst der Bildschirmaufbau, Überlesen führender Leerzeichen und die Eingabe erledigt. Danach beginnt die eigentliche Analyse. Um gleich zu Beginn Fehlerfälle auszuschließen, entscheidet eine Abfrage, ob das erste gefundene Zeichen erlaubt ist, also entweder »a«, »(« oder »)«. Im Fehlerfall wird das Fehlerflag gesetzt. Falls es sich um ein erlaubtes Zeichen handelte, kann in einem WHILE-Konstrukt der Klammerebenenzähler solange inkrementiert werden, wie das nachfolgend eingelesene Zeichen ein »(« ist. Damit steht in dem Zähler die Zahl der öffnenden Klammern. Nun wird abgefragt, ob das von der Klammer verschiedene Zeichen ein »a« ist. Das Fehlerflag zeigt einen Fehler an. Im anderen Fall kann mit dem Zählen der schließenden Klammern begonnen werden. Dazu dient wieder ein WHILE-Konstrukt mit Dekrementierung der Klammerebene und dem Einlesen des nächsten Zeichens.

In der nun letzten Abfrage, ob das nachfolgende Zeichen ein Leerzeichen oder die Klammerebene gleich Null ist, werden die letzten Fehlermöglichkeiten des Wortes abgefangen.

Zum Schluß wird wieder ebenso wie in der rekursiven Lösung, je nachdem, ob ein Fehler festgestellt wurde oder nicht, ein positiver oder negativer Hinweis ausgegeben.

(Max Moser/Harry Paintner/hg)

```
program zeichenfolgen_iterativ (input,output);

{*****************************************}
{***                                   ***}
{*** Iterative Analyse einer Gramatik ***}
{***                                   ***}
{*****************************************}

var zeichen : char;
    fehler  : boolean;
    ebene   : integer;

{***********************}
{***                 ***}
{*** Hauptprogramm ***}
{***                 ***}
{***********************}

begin
   writeln;
   writeln (' Eingabe ');
   readln;
   repeat
      read(zeichen);
   until (zeichen <> ' ');
   fehler := false;
   if (zeichen <> '(') and (zeichen <> 'a')
      then
         fehler := true
      else
 begin
      ebene := 0;
      while (zeichen ='(') do
         begin
            ebene := ebene + 1;
            read (zeichen);
         end;
      if (zeichen <> 'a')
         then
            fehler := true
         else
            begin
               read(zeichen);
               while (zeichen = ')') do
                  begin
                     ebene := ebene - 1;
                     read (zeichen);
                  end;
               if (zeichen <> ' ') or (ebene <> 0) then
                  fehler := true;
            end;
   end;
   writeln;
   if fehler
      then
         writeln('* Fehler *')
      else
         writeln('# O.k. #');
   writeln
   end.
```

Listing 2. Die iterative Umsetzung ist nicht ganz so schön

Rekursive Spielereien

Die Fibonacci-Zahlen sind ein schönes Beispiel, um die Unterschiede zwischen rekursiver und iterativer Programmierung verstehen zu lernen.

Ein lehrreiches Beispiel der rekursiven Programmierung, das heißt der Programmierung von Unterprogrammen, die sich selbst aufrufen, ist ein Lösungsweg zur Berechnung der Fibonacci-Zahlen.

Der im Mittelalter lebende, italienische Mathematiker Leonardo Pisano (zirka 1180 bis 1240), stellte eine Zahlenfolge auf, in der jede Zahl die Summe der beiden vorhergehenden Zahlen darstellt und deren erste Elemente mit 0 und 1 vordefiniert sind.

Daraus ergeben sich folgende Elemente der Folge:

```
0. Element :  0
1. Element :  1
2. Element :  1 =  0 + 1
3. Element :  2 =  1 + 1
4. Element :  3 =  2 + 1
5. Element :  5 =  3 + 2
6. Element :  8 =  5 + 3
7. Element : 13 =  8 + 5
8. Element : 21 = 13 + 8 und so weiter
```

Man kann die Entwicklung auch mit folgender Formel darstellen:

f(i) = f(i-1) + f(1-2)

Es gibt nun zwei Möglichkeiten, die Fibonacci-Zahlen mit dem Computer iterativ und rekursiv zu bestimmen:

Listing 1 zeigt die iterative Lösung mit einer REPEAT-UNTIL-Anweisung. Sie ist zwar ebenso effektiv wie die rekursive Lösung (Listing 2), aber bei weitem nicht so elegant.

Allerdings ist zu beachten, daß bei rekursiver Programmierung der Stack überlaufen kann, wenn man zu große Zahlenwerte benutzt, da alle Daten der Rekursion (Rechenergebnisse, Rücksprungpointer und so weiter) auf dem Stack zwischengespeichert werden.

So wird zum Beispiel das 4. Element der Fibonacci-Folge in folgenden Schritten berechnet (siehe Bild).

(Max Moser/Harry Painter/hg)

```
          FIBONACCI(4)
        = FIBONACCI(3)                              +FIBONACCI(2)
        = FIBONACCI(2)              +FIBONACCI(1)+FIBONACCI(1)+FIBONACCI(0)
        = FIBONACCI(1)+FIBONACCI(0)+1             +1            +0
        = 1           +0            +2
        = 3
```

Bild. Berechnung des 4. Elements der Fibonacci-Folge

```
program fibonacci_iterativ (input,output);
var index, zahl, vletzte, letzte, aktuell : integer;
begin
   repeat
      write ('Bitte geben Sie die Elementnummer der Fibonacci-Folge ein: ');
      readln (zahl);
   Until Zahl >= 0;
   write ('Die Fibonacci-Zahl ist ');
   if zahl <= 2 then
   begin
      case zahl of
         0   : writeln ('0');
         1,2 : writeln ('1');
      end;
   end else
   begin
      vletzte := 1;
      letzte  := 1;
      index   := 2;
      repeat
         aktuell := vletzte + letzte;
         vletzte := letzte;
         letzte  := aktuell;
         index   := index + 1;
      until index = zahl;
      writeln (aktuell);
   end;
end.
```

Listing 1. Die Fibonacci-Zahlen iterativ berechnet

```
program fibonacci_rekursiv (input,output); (*$A-*)
var zahl : integer;
function fibo (wert : integer ) : integer;
begin
   if wert > 1
      then fibo := fibo (wert-1) + fibo (wert-2)
         else if wert = 1
         then fibo := 1
         else fibo := 0
end;

(* Hauptprogramm *)

begin
   repeat
      write ('Bitte geben Sie die Elementnummer der Fibonacci-Folge ein: ');
      readln (zahl);
   until zahl >= 0;
   writeln ('Die Fibonacci-Zahl ist ',fibo (zahl));
end.
```

Listing 2. Die Fibonacci-Zahlen rekursiv berechnet. Die Zeichenfolge »(*$A- *)« schaltet den Compiler auf rekursiven Code um. Viele Compiler (so auch Turbo-Pascal) arbeiten mit so einer Umschaltanweisung.

Numerische Integration nach Simpson

Integralberechnung ist mit dem Computer in Pascal eine feine Sache. Nur die richtige Formel muß man kennen.

Für mathematische Berechnungen sind Computer geradezu prädestiniert, man muß ihnen nur sagen, wie. Mit der Simpson-Formel lassen sich bestimmte Integrale der Form

$$F(x)\Big|_a^b = \int_a^b f(x)dx$$

näherungsweise berechnen.
Die Voraussetzungen hierbei sind:
- a,b sind reelle Zahlen und es gilt a < b.
- y=f(x) sei eine im Intervall [a,b] stetige und differenzierbare Funktion.

Die Lösungsformel lautet:

$$F(a,b,2n) = (h/3) * [f(a) + f(b) + 4 \sum_{i=1}^{n} f(a + (2i-1)h) + 2 \sum_{i=1}^{n-1} f(a + 2ih)]$$

wobei das Intervall [a,b] in 2n Teilintervalle der Breite
h = (b - a)/2n
geteilt wird.
Das folgende Programm berechnet mit Hilfe der Funktion SIMPSON zu

der gegebenen Funktion $f(x) = \exp(-x^2) \int_0^x \exp(t^2)dt$ mit

dem Intervall [0,2]
und der Stellengenauigkeit m=5
die Näherungswerte $F(a,b,2^i)$ mit $1 <= i <= k$.

Die Berechnung wird abgebrochen, wenn folgendes Kriterium erfüllt ist:
$[F(a,b,2^k) - F(a,b,2^{(k-1)})] <= |F(a,b,2^k)| * 0{,}5 * 10^{-m}$.

Programmbeschreibung:

In der Darstellung (Bild 1) sei die Unterteilung des Intervalls, sowie die Gewichtung der Funktionswerte an den Stützpunkten in der Simpson-Formel verdeutlicht:

Das Integral ergibt sich daraus folgendermaßen:
I = (h/3)*(Fktwert(a)+Fktwert(b)+4*(Fktwert(S1)+ Fktwert(S3)+2* Fktwert(S2)

Da bei dieser Intervallunterteilung noch keine großen Ansprüche an die Rechengenauigkeit gestellt werden können, nimmt man eine feinere Intervalleinteilung vor.

Hierbei benutzen wir folgenden Trick:

Die Anzahl der Teilintervalle wird nicht um zwei erhöht, sondern verdoppelt (Teilintervallhalbierung).

Der Schritt der Teilintervallhalbierung und seiner Auswirkung ist in Bild 2 zu sehen.

Dies hat folgende Vorteile:
- Aus der Tatsache, daß die bisherigen Stützpunkte wieder benutzt werden, folgt eine wesentliche Verminderung des Rechenaufwandes.
- Da die Anzahl der Teilintervalle exponentiell ansteigt, konvergiert dieses Verfahren sehr schnell.

Dabei fällt folgendes auf:
- Alle bisherigen Stützpunkte, sowohl die mit der Gewichtung 2 als auch die mit der Gewichtung 4, erhalten bei diesem Schritt die neue Gewichtung 2.
- Die Anzahl der neuen Punkte (also die mit der Gewichtung 4) ist auf das Doppelte gewachsen.
- Die neuen Stützpunkte sind jeweils um die doppelte Intervallbreite voneinander entfernt.

Um diese Erscheinung nochmals zu verdeutlichen, ist in der Darstellung in Bild 3 eine weitere Intervallhalbierung vorgenommen worden.

Durch die stetige Intervallhalbierung wird irgendwann eine genügende Genauigkeit erreicht.

Zur Feststellung dieser Genauigkeit beziehungsweise als Abbruchkriterium für die Unterteilungsschleife eignet sich eine Formel, worin 2^k beziehungsweise 2^{k-1} zwei aufeinanderfolgende Intervallhalbierungen bedeuten. Dieses Abbruchkriterium verwendet schließlich auch M zur Angabe der Zahl der wesentlichen Stellen.

Die bisherigen Erläuterungen des Verfahrens lassen sich einfach in einen Algorithmus umsetzen.

Beim Funktionsaufruf werden eine reelle Funktion f, die untere und obere Integrationsgrenze (--> ug und og, reell), sowie die Rechengenauigkeit m (integer) angegeben. Das Ergebnis der Integration ist natürlich wieder reell.

Damit sieht der Funktionskopf wie folgt aus:

```
FUNCTION SIMPSON (FUNCTION F(X : REAL) : REAL;
                  OG, UG               : REAL;
                  M                    : INTEGER)
                                       : REAL;
```

In der Variablendeklaration sind folgende Bezeichner festgelegt:

```
SUMUO                    : REAL
```

- Summe der Fktwerte der Integrationsgrenzen

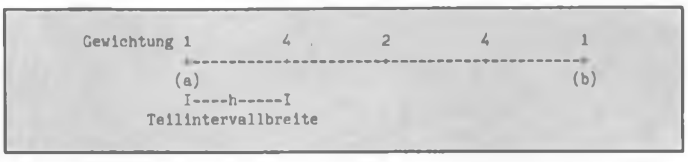


Bild 1. Gewichtung in Teilintervallbreite: zwei wichtige Faktoren für die Simpson-Formel

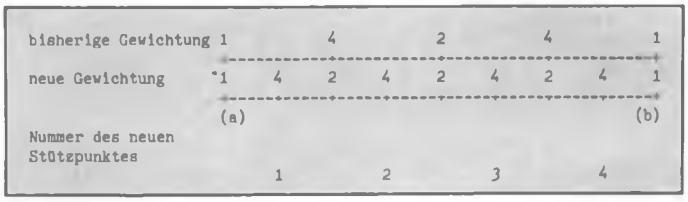


Bild 2. Verkleinern wir die Teilintervalle, so wird das Ergebnis genauer

Bild 3. Jede weitere Halbierung macht das Ergebnis brauchbarer

```
SUMALT                  : REAL
```
- Summe der Fktwerte mit Gewichtung 2 in der Simpson-Formel

```
SUMNEU                  : REAL
```
- neuberechnete Summe der Fktwerte mit Gewichtung 4

```
INTALT, INTNEU          : REAL
```
- letzter und aktueller Integralwert

```
DELTA                   : REAL
```
- aktuelle Teilintervallbreite (h)

```
ZNEUPKT                 : INTEGER
```
- aktuelle Anzahl der neuzuberechnenden Fktwerte

```
COUNT                   : INTEGER
```
- Schleifenzähler zur Summierung der Fktwerte

```
ZEHNHOCHM               : INTEGER
```
- »Konstante«, enthält 10 für Abdruckkriterium

Nun zum eigentlichen Algorithmus. Beim Eintritt in das Programm sind mehrere Größen zu initialisieren:

```
SUMUO                   : = F(UG) + F(OG)
```
- SUMUO wird für den ganzen Fktaufruf so festgelegt.

```
DELTA                   : = (OG - UG)/2
```
- Startwert der Teilintervallbreite

```
ZEHNHOCHM    : = TRUNC(EXP(M*LN(10))) = 10↑M
 SUMALT      : = 0
 INTNEU      : = 0
 SUMNEU      : = 0
```
Startwerte für ersten Durchlauf

```
 ZNEUPKT     : = 1
```
- beim ersten Durchlauf nur 1 neuer Fktwert

Nach der Initialisierung folgt nun die »Integrationsschleife«. Ihre grundsätzliche Struktur stellt ein REPEAT-UNTIL-Konstrukt dar, wobei als Abbruchkriterium die eingangs erwähnte Formel dient.

In der Schleife werden folgende Aufgaben gelöst:

1.) Aufaddieren der neuen Funktionswerte (Abstand: doppelte Teilintervallbreite mit Beginn beim 1. Stützpunkt nach der Untergrenze).

2.) Wenn das Abbruchkriterium nicht erfüllt ist:

a.) Verdoppeln der Anzahl der neu zu berechnenden Stützstellen = ZNEUPKT

b.) Halbieren der Intervallbreite

c.) Sichern des berechneten Integrals für den Vergleich beim nächsten Schleifendurchlauf.

d.) Umbenennen der Funktionswerte mit Gewichtung 4 zu den Funktionswerten mit Gewichtung 2

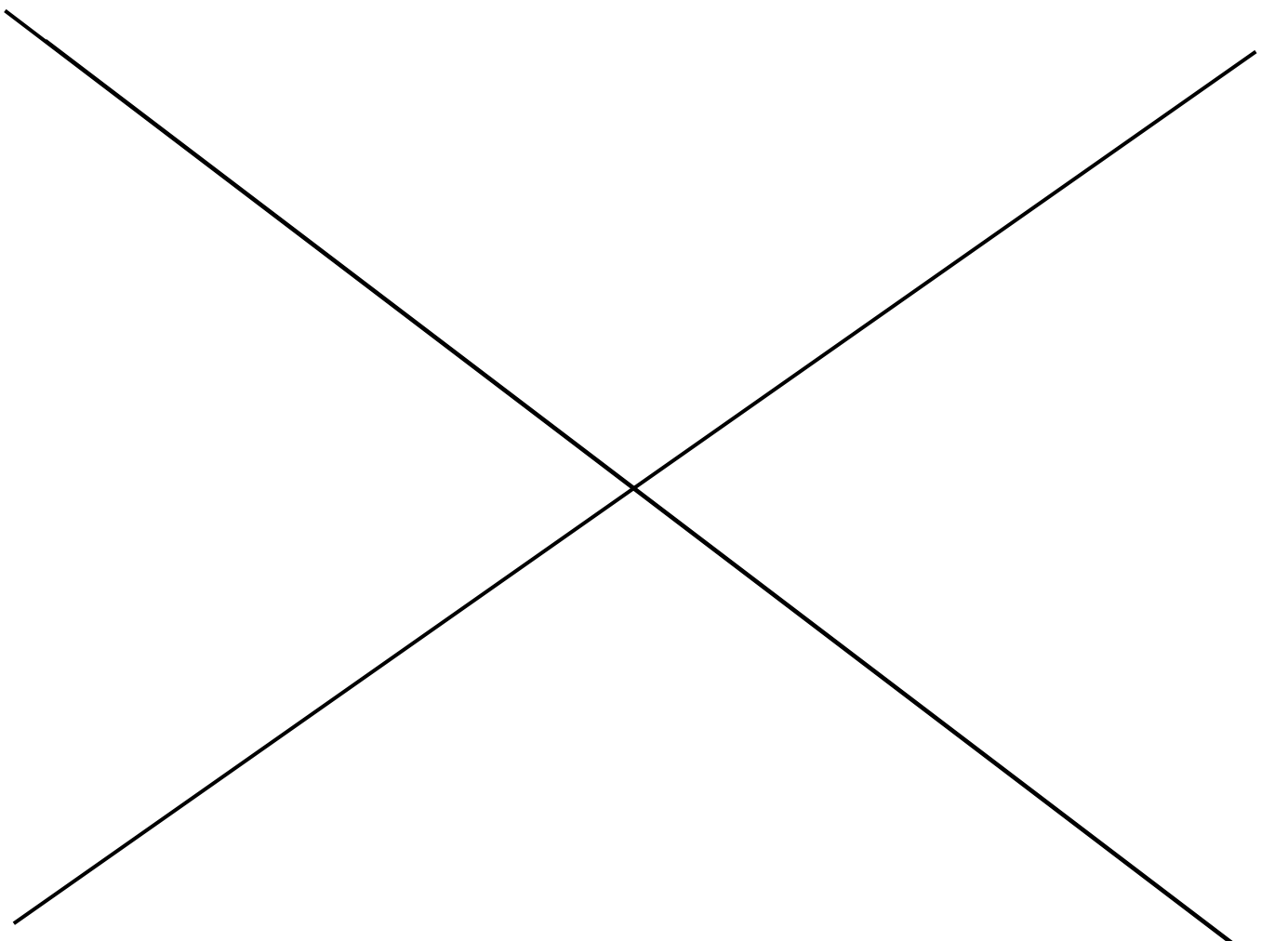
e.) Neuinitialisierung der Summe mit Gewichtung 4 auf 0

In der Funktion SIMPSON sieht die Schleife wie folgt aus:

```
REPEAT
 INTALT := INTNEU;
 SUMALT := SUMALT + SUMNEU;
 SUMNEU := 0;
 FOR COUNT := 1 TO ZNEUPKT DO
SUMNEU := SUMNEU + F(UG+(2*COUNT-1)*DELTA);
 ZNEUPKT := ZNEUPKT*2;
 DELTA := DELTA/2;
UNTIL ⟨Abbruchkriterium erfüllt⟩;
```

Den letzten Teil der Funktion bildet schließlich die Übergabe des beim letzten Durchlauf berechneten Integrals, bei dem das Abbruchkriterium erfüllt wurde, an den Funktionsbezeichner.

Sollte der zur Verfügung stehende Compiler die Übergabe von Funktionen an eine Funktion nicht erlauben (zum Beispiel Turbo-Pascal), so muß man die Funktion SIMPSON im Hauptprogramm realisieren oder man behilft sich, indem man die



Funktion als global vereinbart. Sie muß dann aber vor jedem Programmstart neu eingegeben werden – der Bedienungskomfort sinkt.

Um die Funktion SIMPSON überhaupt nutzen zu können, benötigt man noch ein Rahmenprogramm, das die zusammengesetzte Funktion f(x) im Intervall [a,b] an n+1 Stützstellen auf die wesentliche Stellengenauigkeit m tabelliert.

Die Platzhalter der Daten für Intervallanfang, -ende, Stellengenauigkeit und Stützpunkte wurden als Konstante dargestellt. Sie könnten jedoch auch ohne weiteres vom Programm eingelesen werden. Somit sieht der Programmkopf wie folgt aus:

```
PROGRAM RAHMEN (INPUT;OUTPUT);
 CONST A = 0.0; (*Tabellierungsanfang *)
       B = 2.0; (*Tabellierungsende *)
          N =  40; (*N+1 Stützstellen zur
                     Tabellierung *)
          M =   5; (* Anzahl der signifikanten
                     Stellen zur Integration *)
 VAR  X : REAL; (* Stützstelle als Argument für die
                   Funktion *)
      H : REAL; (* Schrittweite zur Tabellierung *)
      I : 0 .. N; (* Schleifenzähler für die
                     Stützstellen *)
```

Darauf folgt der Funktionsdeklarationsteil. Zum einen wird eine Funktion benötigt, die die angegebene, zusammengesetzte Funktion beinhaltet (FUNCTION FKT). In dieser Funktion ist auch die Integrandenfunktion TEILFKT eingebettet, die dann als aktuelle Parameterfunktion für die Integrationsfunktion SIMPSON verwendet wird. Hier wird mit dem linken Term EXP(-X*X) das Integral von TEILFKT EXP(X*X) in den Grenzen von UG bis OG auf eine Genauigkeit von M wesentlichen Stellen berechnet.

Im abschließenden Hauptprogramm findet nur noch die Berechnung der Funktionswerte an den Stützstellen, sowie die tabellarische Ausgabe der Ergebnisse statt.

(Max Moser/Harry Paintner/hg)

Structogramm der Funktion Simpson

- Initiallsieren der Summe des Anfangs- und Endwerts der Intervallbreite des Integrals,
  der Hilfsvariablen für die neuen bzw. alten Integralwerte und der Hilfsvariablen für die Summe der neuen bzw. alten Stützpunkte des Integrals
  und des Anfangswerts der Anzahl der Teilintervalle des Integrals.
- Teilintervallbreite des Integrals := (Obergrenze-Untergrenze) / 2
- Hilfsvariable für Zehnerpotenz := EXP (Stellengenauigkeit * LN(10))
- Sicherung des letzten Integralwerts für die Abbruchentscheidung
- Sicherung der Summe der letztberechneten Stützpunkte
- Integralvariable auf Null setzen
- Schleife von 1 bis Anzahl Stützpunkte
  - Aufaddieren der neuen Stützpunkte
- Berechnung des neuen Integralwerts
- Verdoppeln der Anzahl der Stützpunkte
- Wiederholen des Blocks, bis die gewünschte Stellengenauigkeit erreicht ist.

Structogramm der Funktion FKT

- Konstante für Integraluntergrenze
- Funktionsaufruf SIMPSON, die Funktion TEILFKT, Integraluntergrenze, Argument und Stellengenauigkeit werden als Aktualparameter übergeben
- Funktionswert FKT := EXP (-x * x) * Funktionswert von SIMPSON

Structogramm des Hauptprogramms

- Konstanten für Intervallanfang, Intervallende, Anzahl Teilintervalle und die gewünschte Stellengenauigkeit
- Ausgabe der Überschrift für die Funktionsliste
- Teilintervallbreite := (Intervallende -Intervallanfang) / Anzahl der Teilintervalle
- Schleife von 0 bis Anzahl der Teilintervalle
  - Argument := Intervallanfang + Schleifenindex * Teilintervallbreite
  - Funktionsaufruf von FKT, Argument und Stellengenauigkeit werden als Aktualparameter übergeben
  - Ausgabe des Arguments und des Funktionswerts in die Ausgabeliste

```
program rahmen (input,output);

{*****************************************************************}
{***                                                           ***}
{*** Rahmenprogramm zur Nacherungsweisen Integration nach Simpson ***}
{***                                                           ***}
{*****************************************************************}

const a = 0.0;
      b = 2.0;
      n =  40;
      m =   5;
var   x,hireal;
      i  :0 .. n;

{*****************************************************************}
{***                                                           ***}
{***     Naherungsweise Integration ueber eine Funktion f(x)   ***}
{***                                                           ***}
{*****************************************************************}

function simpson (function f(x:real):real;
                              ug,og    :real;
                              m        :integer):real;

var susuo,sumalt,sumneu       : real;
    intalt,intneu,delta       : real;
    zneupkt,count,zehnhoche   : integer;

begin
   susuo     := f(ug)+f(og);
   delta     := (og-ug)/2;
   zehnhoche := trunc (exp(m*ln(10)));
   sumalt    := 0;
   sumneu    := 0;
   intneu    := 0;
   zneupkt   := 1;
   repeat
      intalt := intneu;
      sumalt := sumalt + sumneu;
      sumneu := 0;
      for count := 1 to zneupkt do
         sumneu := sumneu + f(ug+(2 * count-1) * delta);
      intneu := delta * (susuo + 4 * sumneu + 2 * sumalt) / 3;
      zneupkt := zneupkt * 2;
      delta   := delta / 2;
   until (abs (intneu-intalt)*zehnhoche <= abs (intneu / 2));
   simpson := intneu;
end; (* of function simpson *)

{*****************************************************************}
{***                                                           ***}
{***                  Zu tabellierende Funktion                ***}
{***                                                           ***}
{*****************************************************************}

function fkt (x:real;m:integer) : real;
   const ug = 0.0;
   function teilfkt (t:real) : real;
   begin
      teilfkt := exp (t*t);
   end; (* of function teilfkt *)
   begin
      fkt := exp (-x * x) * simpson (teilfkt,ug,x,m);
   end; (* of function fkt *)

{*****************************************************************}
{***                                                           ***}
{***                       Hauptprogramm                       ***}
{***                                                           ***}
{*****************************************************************}

begin
   writeln; writeln; writeln;
   writeln ('     ***     x    ***        f(x)         *** ');
   writeln;
   h := (b-a)/n;
   for i := 0 to n do
      begin
         x := a + i * h;
         writeln ('        ',x:12,'       ',fkt(x,m):16);
      end;
   writeln;
end.
```

Listing. Integration nach Simpson

Die Türme von Hanoi

Haben Sie auch Probleme mit der Lösung der Knobelaufgabe »Türme von Hanoi«? Aber benutzen Sie doch einfach Ihren Computer und lassen Sie ihn für sich knobeln.

Wer kennt nicht das Problem der Türme von Hanoi? Die Aufgabe: Es existieren drei Plätze, wobei auf einem der äußeren Standorte ein aus mehreren, immer kleiner werdenden Scheiben bestehender Turm steht. Nun muß dieser Turm auf den anderen äußeren Platz umgelagert werden. Natürlich soll nicht der komplette Turm auf einmal transportiert werden, sondern Scheibe für Scheibe. Bei dieser Umschichtung darf immer nur eine kleine Scheibe auf eine größere Scheibe gelegt werden.

An dieser Knobelaufgabe haben sich schon große Geister versucht und auch sehr lange darüber gegrübelt. Wenn nur drei Scheiben auf dem Turm liegen, ist es ja noch relativ einfach herauszufinden. Haben Sie dagegen einen Turm mit vielleicht 20 Scheiben, so ist es so gut wie nicht mehr zu schaffen (es sei denn, Sie hätten einige Jahre Zeit dafür).

Die Prozedur »hanoi« (Listing) nimmt Ihnen nun die Denkarbeit vollständig ab. Sie müssen nur die Anzahl der Scheiben des Turmes eingeben, und schon erscheint die Ablegevorschrift auf dem Bildschirm.

Versuchen Sie aber ruhig zuerst einmal, das Problem mit drei, vier oder fünf Scheiben »zu Fuß« anzugehen, bevor Sie sich die Lösung vom Computer zeigen lassen.

(Dieter Mayer/hi)

```
PROGRAM hanoi(INPUT,OUTPUT);
VAR
   n,i:real;
   etagenzahl:0..MAXINT;

   PROCEDURE verlagere(hoehe,von,zu,ueber:integer);
         PROCEDURE tragt(runter,rauf:integer);
                BEGIN
                    write(runter:1,'>',rauf,'  ');
                    i:=i+4;
                    n:=n+1;
                    IF i>=80 THEN BEGIN
                      i:=0;
                    END (*end if*)
                END;
   BEGIN (*verlagere*)
         IF hoehe>0 THEN BEGIN
            verlagere(hoehe-1,von,ueber,zu);
            tragt(von,zu);
            verlagere(hoehe-1,ueber,zu,von);
         END;   (*end if*)
   END;  (*end verlagere*)

BEGIN
      write('Wegen Verstaendigungs');
      writeln('schwierigkeiten in Hanoi muss der ');
      write('Turm von Hanoi-Nord nach Hanoi-Sued ');
      writeln('verlegt werden.');
      writeln('Ein Zwischenlager gibt es in Hanoi-Mitte.');
      writeln('Beim Verlegen ist zu beachten:');
      writeln('-nur die jeweils oberste Etage kann ');
      write(' von einem Lager in ein anderes');
      writeln(' transportiert werden');
      write('-eine Etage darf nicht auf ');
      writeln('eine kleinere gelegt werden');
      writeln;
      write('Bitte die Anzahl der Etagen angeben: ');
      readln(etagenzahl);
      i:=0;
      n:=0;
      verlagere(etagenzahl,1,2,3);
      writeln;
      writeln('Anzahl der Umlagerungen: ',n:5:0);

END.
```

Zahlen-Tabellen

Ein nützliches Programm, das Ihnen auf einen Blick Zahlen und deren Quadrat- oder Kubik-Funktionen übersichtlich aufzeigt. Es gibt oft Situationen, in denen diese kleine Anwendung recht nützlich ist.

Diese kleine Routine zeigt Ihnen die Quadrat-, Kubik-, »Hoch 4«- und »Wurzel«-Werte einer Zahl in einer übersichtlichen Tabelle. So können Sie sich auf einfache Art eine Liste erstellen, die Ihnen die gebräuchlichsten Werte auf einen Blick liefert. Allerdings werden nur Zahlen bis 181 richtig verarbeitet (dies hängt mit dem Format der Real-Zahlen zusammen).

Zum Programm

Die Prozedur »clrscr« ist MS-DOS-spezifisch. Man kann sie auch weglassen, da hiermit nur der Bildschirm gelöscht wird. Wollen Sie Zahlen ausgeben, die größer als 21 sind, so ist nur die Bedingung in der Until-Schleife auf einen höheren Wert als 21 zu setzen (maximal 181), oder das Format der Zahl in der Variablendefinition in Real umzuwandeln, und die Ausgabeformatierung »zahl« in der Repeat-Schleife durch »write(zahl:4:0,...« zu ändern. (Dieter Mayer/hi)

```
PROGRAM tabelle;
VAR
   zahl:integer;
   wurzel,quadrat,kubik,hoch4:real;
BEGIN
  zahl:=0;
  clrscr;
  write('Zahl:        Quadrat:        Kubik');
  writeln(':        Hoch 4:        Wurzel:');
    repeat
      quadrat:=zahl*zahl;
      kubik:=quadrat*zahl;
      hoch4:=quadrat*quadrat;
      wurzel:=sqrt(zahl);
      write(zahl:2,'          ',quadrat:5:0);
      write('            ',kubik:5:0);
      write('        ',hoch4:7:0);
      writeln('       ',wurzel:10:7);
      zahl:=zahl+1;
    until zahl>21
END.
```